Habilitation à Diriger des Recherches

Université Grenoble Alpes

Informatique

Formal Model Driven Engineering

Akram Idani

VASCO
Laboratoire d'**I**nformatique de **G**renoble (**LIG**)

Date: 26 Mai 2023

Rapporteurs :

Catherine Dubois	ENSIIE, Evry	Professeure
Pierre-Yves Schobbens	Université de Namur	Professeur
Virginie Wiels	ONERA, Toulouse	Directrice de recherche

Examinateurs :

Sophie Dupuy-Chessa	Université Grenoble Alpes	Professeure
Régine Laleau	Université Paris-Est Créteil	Professeure
Michael Leuschel	Université de Düsseldorf	Professeur

BibTeX

```
@PhdThesis{hdr-idani,
    author = {Akram Idani},
    title = {Formal Model Driven Engineering},
    school = {Université Grenoble Alpes},
    year = {2023},
    type = {Habilitation à Diriger des Recherches (HDR)},
    address = {LIG},
}
```

باسم الله الرحمان الرحيم

« Le secret. . . de l'expression et du message, consiste à communiquer des idées. Sans transmission de la pensée, le langage n'est qu'une terre morte. »
Ibn Khaldoun (Prolégomènes, 1377)

à mes ancêtres et leur descendance . . .

à mes parents et mes enfants . . .

à mon frère et à ma soeur . . .

. . .

à mon étoile, à la belle sirène et aux airs de guitare !

Remerciements

Toutes les personnes m'ayant permis de mener à bien ce travail sont assurées de ma gratitude : « *Grâce à vous ce qui n'était qu'une question de recherche est devenu une passion* ».

⋆ ⋆ ⋆

À mon collègue, chef d'équipe et directeur de thèse, Yves Ledru : « *Je t'adresse, en signe de reconnaissance, mes plus vifs remerciements pour ta gentillesse et ton soutien. Tu as manifesté à mon égard de bienveillants conseils tout au long de mon parcours* ».

À mon collègue, German Vega, Ingénieur de Recherche au CNRS : « *J'exprime mes sincères remerciements pour tous les échanges qu'on a pu avoir et qui ont permis d'élaborer le cadre applicatif de mes travaux. Sans toi les outils B4MSecure et Meeduse seraient restés à un état embryonnaire. J'espère que tu trouveras ici l'expression de ma gratitude et de ma grande estime pour ta personne, ainsi que ma profonde considération pour ta compétence professionnelle* ».

⋆ ⋆ ⋆

Il n'est pas aisé, en quelques lignes, de remercier toutes les personnes ayant permis à ce modeste travail de prendre forme. C'est grâce aux collègues, étudiants, amis et proches que ce travail est ce qu'il est aujourd'hui. Ce mémoire est une synthèse d'une dizaine d'années de recherches durant lesquelles j'ai fait des rencontres riches qui ont forgé ma pensée scientifique et qui ont contribué à la maturité de mes réflexions.

C'est avec reconnaissance et sincérité que je remercie les membres du jury pour leur travail d'évaluation. Je remercie vivement Catherine Dubois, Virginie Wiels et Pierre-Yves Schobbens, pour leurs judicieux commentaires et leurs remarques et critiques très constructives. Je remercie aussi Sophie Dupuy-Chessa, Régine Laleau et Michael Leuschel qui m'ont fait l'honneur d'examiner ce travail : « *Votre regard avisé sur les nombreux sujets traités dans ce mémoire est fort utile pour moi car il constitue un guide pour mes recherches futures* ».

Merci particulièrement à mes collègues du LIG, de l'UGA et de Grenoble INP. Je ne saurais les citer tous en cette page. Grâce à leur bonne humeur et à l'ambiance chaleureuse j'ai agréablement effectué mon travail et mené une carrière d'enseignant-chercheur épanouissante.

Il ne fait aucun doute que mes travaux se sont enrichis au fil des années des nombreux échanges que j'ai eus avec mes collègues, mais aussi avec les jeunes étudiants-chercheurs en master et en thèse. Je tiens à les remercier très sincèrement pour avoir contribué à mes recherches. Je témoigne ici de leur mérite et je leur souhaite un avenir plein de succès.

Finalement, je tiens à souligner que ce travail est aussi le fruit d'encouragements et de bienveillance de la part de mes ami(e)s : « *Votre bonté sincère et générosité inconditionnelle font de vous des personnes exceptionnelles.* ».

⋆ ⋆ ⋆

Outline

Introduction

La notion d'abstrait est comprise comme ce qui est flou, pas clair,
pas ou peu compréhensible, peu ou pas accessible.
En fait, l'abstrait est ce qui est virtuel, conceptualisé.
C'est l'univers mental, les pensées, les idées, les productions
de cette aptitude à faire des concepts à partir
de mes sensations, de mes émotions, ou de mes intuitions.
L. Martinez

Preamble

My research works are dedicated to the integration of two well known paradigms: Formal Methods (FM) and Model-Driven Engineering (MDE). This integration is called Formal MDE (FMDE) all along the current document. In fact, several works have been already done in order to strengthen the MDE paradigm with formal reasoning, and therefore make it more viable as far as safety and security concerns have to be addressed. When taken separately, these works provide a partial coverage of MDE, but when combined they can address a wide range of models and languages. During the last decade, I investigated two directions in which the FMDE paradigm proved its value: (*i*) Model-Driven Security (MDS), and (*ii*) Domain-Specific Languages (DSLs). Under the MDE umbrella, both the MDS and DSL communities advocate for the use of models throughout the development process, providing solutions to the validation problem ('*do the right system*'). Nonetheless, the verification problem ('*do the system right*') is still a major challenge, perhaps because formal reasoning (*i.e.* model-checking and/or theorem proving) was not apart of the MDE initiative. To be pragmatic my contributions build on well-established notations: mainly UML and B, and − at a smaller scale − BPMN, CSP, Z and Petri-Nets. Besides, the obtained results can be inspiring and, in my opinion, should be extended with other approved (semi-)formal languages, which would confer to FMDE a broader spectrum. The next two sections (A. and B.) summarize for every research direction (respectively MDS and DSLs) the challenges that guided my works, and give an overview of my contributions and publications in the field.

A. Model-Driven Security

A.1. Challenges

Computer security often refers to hackers or intruders, who are persons with high technical skills and whose intention is to exploit security breaches in order to get an illegal access to a system. However, in reality the greatest threats come from inside the system, *i.e.* from trusted users who are already granted a legal access. This kind of threat is called "insider attack" in cyber-security and it is known to be difficult to tackle (Probst et al., 2010). Studies done by IBM X-Force Research in the cyber-security landscape state that: "*In 2015, 60 percent of all attacks were carried out by insiders* [...] *and they resulted in substantial financial and reputational losses*". The problem is beyond the access control frontier since it includes unpredictable human behaviours. To deal with these threats, existing industrial, academic and government studies (Kont et al., 2018, Greitzer, 2019, Homoliak et al., 2019) elaborate human profiles and advocate for the use of surveillance systems. Without being exhaustive, some of these profiles are:

- Curious persons who, without a malicious intention but without self-control too, get access to sensitive data or do some actions that are in contradiction with the company rules.
- Super-heroes who, in order to fix a problem or help someone, bypass the company policies believing that it may be useful or simply be approved.
- Audacious persons who, in order to prove their strength (most of the time to them-selves), secretly get access to private information available in the company network.

Other profiles are established in the literature, like machiavellian, greedy, disgruntled, opportunistic, etc. Unfortunately, the eventuality of a breach of trust is difficult to predict in advance based on human-centric factors. On the one hand there is no certainty about a possible acting out, and on the other hand people surveillance must comply with privacy legislation, which makes it almost ineffective. Nonetheless, Information Systems (IS) together with their business logic and processes, provide useful knowledge allowing one to deal with the insider threat problem. In fact, based on the aforementioned studies it can be observed that insiders often do not have high computer skills (contrary to intruders), but they have a fine-grained knowledge about the IS procedures. The latter are mostly well-established and already protected via access control mechanisms. Hence, by being able to answer the question "*who has access to sensitive data and what kind of access is given?*", one cannot claim that the system is secure enough. The good question should be: "*is the user able to run a sequence of actions that may bring him from a prohibition to an authorization?*". The first question refers to static concerns, and it is widely addressed in Model-Driven Security (MDS) thanks to several access control models (SecureUML, UMLSec, etc). However, the second question remains open in MDS because it refers to behavioural features and the reachability of unwanted situations granting to the user misappropriated privileges.

A.2. Contributions

I contributed toward MDS through a FMDE approach covering structural and behavioural features of secure IS, and addressing their validation against insider threats. The proposed security-by-design technique builds on the strengths of UML, SecureUML (a UML profile for access control) and formal languages (B (Abrial, 1996) and Z). The aim is to combine graphical views and automated reasoning during the IS development. This research began in 2008 in the context of the ANR-Selkis project[1] and has been the subject of several PhD and master projects that I (co-)supervised. To summarize, my contributions can be separated in two major axes: Modeling, and Analysis.

A.2.1 Axis 1: Modeling

Separation of concerns is a core technique in MDS (Basin et al., 2011), since MDS means "*applying the concepts behind model-driven software development to security*". It has been proposed to master the complexity of the IS by distinguishing its functional concerns (data model and the associated business logic) from high-level ACIT properties (Availability, Confidentiality, Integrity, Traceability). In order to guarantee a security-by-design approach in which the MDS benefits from an automated formal support, my works first apply UML and SecureUML to represent both functional and security concerns, and then suggest their translation into B and Z. I co-supervised two PhD students on this topic: M.-A. Labiadh and N. Qamar. The research work of M.-A. Labiadh was built on the B method, and proposed configurable transformations allowing one to mix several transformation approaches. Regarding the work of N. Qamar, it applied a direct translation using RoZ, a tool by Ledru (2006) whose aim is to extract Z specifications from UML class diagrams. Both techniques are complementary and provide together a good coverage of MDS. In the remainder I will focus on the usage of the B method, mainly because it is a backbone of my current and further works in FMDE.

Functional models. The extraction of B specifications from UML models is not a new topic; it was widely investigated between 2002 and 2006 resulting in a rich state of the art. However, an exhaustive analysis of the existing approaches show that each kind of UML-to-B mapping has its specific objectives and characteristics and, except iUML-B by Snook and Butler (2006a), a major issue is that tools are either obsolete or not publicly available. Thus, to be applied in MDS efforts have been done to polish, adapt, extend and combine those works in a usable framework. The proposal done in the work of M.-A. Labiadh is to make the UML-to-B mappings as parametric as possible, such that the user can easily manage the transformation rules and experiment various solutions.

Security models. Regarding the translation of SecureUML, it has been built on formal definitions of the Role-Based Access Control model (RBAC). RBAC is supported by several software products like popular commercial database management systems (*e.g.* Oracle, Sybase) or

[1] ANR-08-SEGI-018, 20012–2018.

webservers (*e.g.* JBoss). The available implementations of this model act like a filter which intercepts a user request to a resource in order to permit or deny the access to associated functional actions (*e.g.* transactions on databases, file operations, etc). The thesis of M.-A. Labiadh followed a similar principle, but at a modeling stage, leading to the extraction of RBAC filters written in B, and in which permissions/prohibitions are defined as security guards.

Business process models. Another important pillar of an IS, in addition to its functional and security concerns, is to perform established processes that are followed by the stakeholders during their working activities. A business process model represents a set of steps undertaken for a specific purpose in which intrinsically operations concerning the IS data (like reading, modification, etc.) and responsibilities for performing tasks are defined. The three concerns (functional models, security policies and business processes) are important; their alignment is useful to address several inconsistencies, such as when the access control policy gives more permissions than the actions required by the business process. In this case following the process may hide several authorizations giving the impression that some bad actions are not possible while they can still be done from outside the process, which is a typical example of insider attacks. This topic was addressed in the PhD of S. Chehida, and by two master 2 students during their trainees. It also led to an established collaboration with the SIGMA team from the LIG lab, with local fundings. The objective is to align the three models and provide a technique to verify the correctness of this alignment using B.

A.2.2 Axis 2: Analysis

A direct consequence of the separation of concerns principle is that functional and security models are often validated separately. Existing works in MDS are therefore stateless and they mostly validate security policies statically without taking into account the dynamic evolution of the IS. Nonetheless, authorized actions often lead to evolutions of the functional state, which may influence the security behaviour and favour insider threats. A well known insider attack which was possible due to evolutions of the functional state is that of 'Société Générale', perpetrated by Jérôme Kerviel. This attack resulted in a net loss of $7.2 billion to the bank[2]. The insider circumvented internal security mechanisms to place more than $70 billion in secret, unauthorized derivatives trades. Through authorized actions, he was able to cover up operations he has made on the market by introducing into the functional system fictive offsetting inverse operations, so that the unauthorized trades were not detected. Dynamic analysis based on reachability properties is therefore crucial because it would establish that a system evolves as expected and that unwanted situations are not possible.

By using B and its composition mechanism, my works provide a way to take into account the intertwining of security policies and functional concerns of the IS. The underlying analysis techniques are intended to verify whether a given targeted functional state is reachable, and eliminate threats by proving that a given unwanted state is unreachable. Since an (un)wanted

[2] The New York Times. French Bank Says Rogue Trader Lost $7 Billion. January 2008.

state can be formally characterized with a predicate P, then one can assume that insider threats start from an initial normal situation where P is false, and may occur if it is possible to exhibit a sequence of B operations, whose security guards are true, and leading to a state where P holds. The solutions I investigated build on two strategies: forward search and backward search.

Forward search. Forward search is a classical approach that is often done thanks to model-checking techniques, but it is more efficient for relatively small applications. As this is not always possible in the case of IS, a model-checker may require some guidance. To this purpose I investigated, on the one hand, process-based search applying CSP||B (Butler and Leuschel, 2005); and on the other hand, an original optimisation technique built on a variant of the ant colony algorithm (Merkle and Middendorf, 2006). This part of the work was developed in collaboration with students from the master 2 MoSIG, and two ENSIMAG students during the teaching unit "Initiation à la recherche en laboratoire".

Backward search. Backward search starts from the unwanted state and tries to go back to a normal situation. In other words it starts by qualifying the attack (P) and then looks at the authorized operations, called critical, that reach P from $\neg P$. Hence, rather than looking for one unwanted state (like in forward search), the technique considers that every precondition of every critical operation is itself an unwanted state. This topic was addressed during the thesis of A. Radhouani, leading to the development of the GenISIS tool (Generator of Insider Scenarios in Information Systems). The tool provides a symbolic backward search applying both theorem proving and constraint solving approaches.

A.3. Main publications

Akram Idani and Mario Cortes Cornax. Towards a model driven formal approach for merging data, access control and business processes. In *2nd International Workshop on Security for and by Model-Driven Engineering (SecureMDE). MODELS'20 Companion Proceedings*, pages 57:1–57:5. ACM, 2020. doi: 10.1145/3417990.3420046.

Yves Ledru, Akram Idani, Rahma Ben Ayed, Abderrahim Ait Wakrime, and Philippe Bon. A separation of concerns approach for the verified modelling of railway signalling rules. In *Third International Conference on Reliability, Safety, and Security of Railway Systems - Modelling, Analysis, Verification, and Certification (RSSRail)*, volume 11495 of *LNCS*, pages 173–190. Springer, 2019. doi: 10.1007/978-3-030-18744-6_11.

Akram Idani. Model driven secure web applications: the SeWAT platform. In *5th European Conference on the Engineering of Computer-Based Systems, ECBS 2017*, pages 3:1–3:9. ACM, 2017. doi: 10.1145/3123779.3123800.

Salim Chehida, Akram Idani, Yves Ledru, and Mustapha Kamel Rahmouni. Combining UML and B for the specification and validation of RBAC policies in business process activities. In

10th IEEE International Conference on Research Challenges in Information Science, RCIS, pages 1–12. IEEE, 2016. doi: 10.1109/RCIS.2016.7549284.

Amira Radhouani, Akram Idani, Yves Ledru, and Narjes Ben Rajeb. Symbolic Search of Insider Attack Scenarios from a Formal Information System Modeling. *LNCS Transactions on Petri Nets and Other Models of Concurrency*, 10:131–152, 2015. doi: 10.1007/978-3-662-48650-4_7.

Yves Ledru, Akram Idani, Jérémy Milhau, Nafees Qamar, Régine Laleau, Jean-Luc Richier, and Mohamed-Amine Labiadh. Validation of IS security policies featuring authorisation constraints. *International Journal of Information System Modeling and Design (IJISMD)*, 6 (1), 2015a.

Akram Idani and Yves Ledru. B for Modeling Secure Information Systems - The B4MSecure Platform. In *17th International Conference on Formal Engineering Methods, ICFEM*, volume 9407 of *LNCS*, pages 312–318. Springer, 2015. doi: 10.1007/978-3-319-25423-4_20.

Yves Ledru, Akram Idani, and Jean-Luc Richier. Validation of a security policy by the test of its formal B specification - A case study. In *3rd IEEE/ACM FME Workshop on Formal Methods in Software Engineering, FormaliSE'15*, pages 6–12. IEEE Computer Society, 2015b. doi: 10.1109/FormaliSE.2015.9.

Amira Radhouani, Akram Idani, Yves Ledru, and Narjes Ben Rajeb. Extraction of insider attack scenarios from a formal Information System Modeling. In *5th International Workshop on Formal Methods for Security (FMS)*, 2014.

Jérémy Milhau, Akram Idani, Régine Laleau, Mohamed-Amine Labiadh, Yves Ledru, and Marc Frappier. Combining UML, ASTD and B for the formal specification of an access control filter. *International Journal on Innovations in Systems and Software Engineering (ISSE)*, 7 (4):303–313, 2011. doi: 10.1007/s11334-011-0166-z.

Yves Ledru, Nafees Qamar, Akram Idani, Jean-Luc Richier, and Mohamed-Amine Labiadh. Validation of Security Policies by the Animation of Z Specifications. In *Proceedings of the 16th ACM Symposium on Access Control Models and Technologies*, SACMAT'11, New York, NY, USA, 2011a. ACM.

Nafees Qamar, Yves Ledru, and Akram Idani. Evaluating RBAC supported techniques and their validation and verification. In *6th International Conference on Availability, Reliability and Security, ARES'11*, pages 734–739. IEEE Computer Society, 2011a. doi: 10.1109/ARES.2011.112.

Nafees Qamar, Yves Ledru, and Akram Idani. Validation of security-design models using Z. In *13th International Conference on Formal Engineering Methods, ICFEM'11*, volume 6991 of *LNCS*, pages 259–274. Springer, 2011b. doi: 10.1007/978-3-642-24559-6_19.

Yves Ledru, Akram Idani, Jérémy Milhau, Nafees Qamar, Régine Laleau, Jean-Luc Richier, and Mohamed-Amine Labiadh. Taking into account functional models in the validation of IS security policies. In *CAiSE Workshops*, volume 83 of *LNBIP*. Springer, 2011b.

B. Domain Specific Languages

B.1. Challenges

Domain-Specific Languages (DSLs) have gained significant attention in industry and academy with an increasing number of tools. Their advantage is their ability to offer substantial gains in expressiveness and ease of use compared with general purpose languages. Tools for handling DSLs can be classified in two major currents: (*i*) meta-compilers (Mandell and Estrin, 1966) (named also compiler-compiler) dedicated to the construction of compilers, translators, and interpreters, and (*ii*) language workbenches (Fowler, 2005), designed to build software through a rich environment of multiple, but integrated, domain-centric notations. The latter have received a lot of interest by the MDE community leading to a plethora of approaches with several perspectives: design, execution, debugging, code generation, etc. One can refer to Kosar et al. (2016) for a survey about tools that are referenced from 2006 to 2012, and to Iung et al. (2020) for the period between 2012 and 2019. These studies can help engineers to choose the best DSL option for a specific context and select the ones that fulfill their requirements.

The aforementioned surveys show that DSL tools have reached a good level of maturity, but they also show a major limitation of these tools, the lack of formal reasoning. Kosar et al. (2016) observed that only 5.7% of primary studies applied a formal analysis approach, and mentions that "*there is an urgent need in DSL research for identifying the reasons for lack of using formal methods within domain analysis and possible solutions for improvement*". One possible reason, as discussed in (Bryant et al., 2011), could be that the syntactical description of DSLs is often straightforward, but specifying the semantics is much more harder. This would explain why the semantics are often left toward "*other less than desirable means*". Obviously, for large scale projects formal methods are not as widespread, because of the overhead they may create during the development activities; nonetheless, several specialized conferences have presented a lot of approaches over the years where formal methods found their way in MDE techniques, and vice-versa. One can refer to the MoDeVVa Workshop (Model Driven Engineering, Verification and Validation) collocated with the Models conference, and also to the international conference on integrated formal methods (IFM), etc.

Unfortunately, in spite of the existence of the required material in order to provide a valuable formal approach for DSLs, not only the integration attempts remain poor, but also the applications of existing approaches remain limited to illustrative examples without going further towards realistic safety-critical requirements. This statement is based on the recent study of Iung et al. (2020) because it does not report on a better situation even several years after (Bryant et al., 2011) and (Kosar et al., 2016). In fact, Iung et al. (2020) refer to testing as 'the'

verification feature of language workbenches. Not only the formal dimension is completely absent, but also the mapping shows that among the fifty-nine discussed tools only nine (15%) provide supports for testing. This important shortcoming of existing language workbenches weakens their applicability, especially for safety-critical systems. In these systems correctness is a strong requirement and it is often addressed by the application of formal methods.

B.2. Contributions

The aim of my research works in the context of DSLs is to circumvent the lack of formal reasoning in language workbenches. During the last three years I devoted a great deal of effort-through this topic, leading to several publications and novel research directions. The achieved work is materialized by the Meeduse tool (Idani, A., 2020b, Idani, A. et al., 2020), which is today the only available language workbench that allows one to build and execute proved DSLs. The tool won three awards: "best verification" and "audience award" at the Transformation Tool Contest (Idani, A. et al., 2019c) and "best presentation" at the 2nd ACM Symposium on Software Engineering (Idani, A., 2021). The tool was also demonstrated during an invited keynote at the MoDeVVa'21 workshop (collocated with the Models'21 conference). Furthermore, several realistic applications have been done with Meeduse, showing how useful is a FMDE approach in the field of DSLs, and meaning also that the approach should be investigated further in the future.

B.2.1. Applications

Railway systems. Most modeling tools (*e.g.* SafeCap, RailAid, etc) are not built on a formally defined DSL; and most formal solutions are not often supported by domain-specific modeling tools. The application of Meeduse to the railway field showed its strength to mix both aspects in the same tool. This work was funded by the NExTRegio project of IRT Railenium and supported by SNCF Réseau. The intention was to deal with the analysis of railway signalling systems based on emergent train automation solutions, especially the European Rail Traffic Management System (ERTMS) and the underlying Train Control System (called ETCS). There are three levels of ERTMS/ETCS which differ by the used equipments and the operating mode. The first two levels are already operational, and the third one is still in design and experimentation phases: it aims at replacing signalling systems with a global European GPS-based solution for the acquisition of train positions. This collaboration led to a graphical DSL (Idani, A. et al., 2019b,a) equipped with formal semantics, which safely assists domain experts while defining the domain models and simulating the underlying operating rules. This work provided several perspectives that are currently under study in the PhD thesis of A. Yar.

Model transformations. Model transformation is a core concept in MDE. It is aiming at automating the extraction of platform specific representations and/or executable artifacts from high level models. Despite the advances in this field, the Verification & Validation (V&V) of model transformations still remains scattered, and perspectives on the subject are still open. An

FMDE approach for model transformations makes sense as soon as critical concerns have to be taken into account. Meeduse was used to participate to the 2019 edition of the transformation tool contest (TTC'19). The call for solutions was about a case study that is well-known in safety-critical systems (even if it is an academic one): the transformation of Truth Tables (TT) into Binary Decision Diagrams (BDD). The proposed solution (Idani, A. et al., 2019c) was carried out in a collaboration with M. Leuschel from the University of Düsseldorf and G. Vega from the LIG lab. My major observation during the contest is that among the seven participants, Meeduse was the only attempt that addressed V&V; the other solutions addressed flexibility, performance and optimality. Meeduse was not only used to verify the model transformation, but also to execute it, which is its novelty in comparison with the discussed approaches. However, the downside is its low performance when executing the transformation of huge models (millions of entries). Nonetheless, as the transformation is written in B, proved and attested for middle size models, it can therefore be used as a reference specification from which one can build a low-code model transformation solution.

XML standards. XML is used across a lot of domains because it favours readability (thanks to its structuring features) and interoperability between platforms (thanks to the availability of parsers). Several communities have developed XML-based exchange standards to structure domain concepts and improve the interoperability of the growing number of computer applications that use these domain concepts. In the railway field for example, one can refer to RailML (Railway Markup Language, an open XML based data exchange format for data interoperability of railway applications). An XML file is a DSL, whose static semantics (well-formedness rules) are established via an XML schema, which is itself a meta-model in the MDE jargon. Dealing with XML documents opens a broad spectrum of possibilities to FMDE, since a formal approach can be applied to any domain standard approved by experts. Having this argument, I applied Meeduse to PNML (Petri-Net Markup Language), the international standard ISO/IEC 15909 for Petri-nets. PNML provides an agreed-on interchange format that is compliant with a formal definition of Petri-nets and is managed by EMF-based platforms such as PNML Framework and The ePNK. This application led to a new tool called MeeNET (Idani, A., 2022, 2020a) in which a PNML file can be executed, debugged, verified and translated into an implementation.

B.3. Research directions

The current applications of Meeduse and its underlying FMDE approach show the practical side of my contributions to the field of DSLs. Notwithstanding, several research directions are still under investigation.

Reusing specifications. Several companies, such as Siemens Transport (Essamé, 2004) and Clearsy (Pouzancre, 2003), have an established software development process that follows the B method: B specifications are written from a requirements document, and then refined until

the production of executable programs. However, one of the major difficulties with formal specifications is their poor readability, which prevents user involvement during the validation activities. In (The Standish Group, 2008) it is clearly mentioned that "*lack of user involvement traditionally has been the number one reason for project failure*". Furthermore, as stated by Bjørner (2010) "*before we can formulate requirements, we must understand the* [*application*] *domain*", meaning that domain specific representations are required before starting to think about formal models. In a pragmatic approach, these representations should be provided by the domain expert who has a greater knowledge of the application domain than the formal methods engineer. For these reasons I advocate for the use of DSLs as a way to validate B specifications. Since Meeduse defines the semantics of a given DSL using B, the proposal is to establish a proved linkage, also defined in B, between the provided B specifications and the DSL semantics. This question motivates the ongoing thesis of A. Yar, that is dedicated to the verification and validation of ERTMS/ETCS. On the one hand, one can find many existing formal models of ERTMS/ETCS. For example, the ABZ'2018 conference published several papers and artifacts that provide proved B specifications of ERTMS/ETCS. On the other hand, standard notations such as RailML (RailML, 2018) and Rail Topo Model (International Union of Railways (UIC), 2016), for the design of railway infrastructures are not formally defined. RailML is an XML-based language, and Rail Topo Model is defined via a UML meta-model; and hence, both standards can be seen as DSLs. The question is therefore: "*how to make the bridge between the existing B artifacts, that are already proved correct, and a formal definition of standardized railway notations*?".

DSL Refinements and Models@Runtime. Meeduse links two technological spaces: that of the B method and that of MDE. Technically it connects ProB (Leuschel and Butler, 2008), a powerful animator and model-checker, to EMF[3] (Steinberg et al., 2009), one of the most popular frameworks for building DSLs. This connection allows one to take benefit of the cross contributions of these technological spaces in order to address several interesting challenges for both DSLs and the B method. In this context I investigated two directions: incremental development of DSLs through refinements and the usage of the corresponding formal models at run-time. The first direction builds on the refinement principle of the B method, so that one can incrementally create a DSL, or provide several versions of the same DSL without breaking the global safety properties of the application domain. Regarding the usage of models at run-time, it exploits the pre-established notification mechanism of EMF. This mechanism is responsible for a great deal of the power offered to applications running EMF run-time (Steinberg et al., 2008). This second direction is inspired by (Körner et al., 2019) where the authors proposed to embed a model checker and animator into applications in order to use the formal models themselves at run-time. The approach assumes that a tool like an animator or model checker is able to compute all state transitions and proposes to implement the software system on top of programs that execute in background the formal model by choosing traversing transitions. The Meeduse approach goes a step further by reducing the effort required for writing the aforemen-

[3] EMF: Eclipse Modeling Framework

tioned programs that interact with the animator. This topic has been the subject of a master's project (M2 Génie Industriel) and was supported by DomoSûr, a local project funded by the LIG lab. The project applies Meeduse to instrument an executable smart-home solution that is proved correct. Furthermore, this approach has been applied as a proof of concept in the smart-buildings field to provide a lightweight development of outbreak prevention strategies (Idani, A., 2021).

A pragmatic embedding of B within modelling languages. There exists a plethora of approaches (including our works in MDS) that extract B specifications from modeling languages (UML, BPMN, Petri-Nets, etc). Overall, they share common motivations: (*i*) give a precise definition to the input formalism, and (*ii*) apply the B tools to address the verification tasks. Nevertheless, they also share a common limitation: the extraction process and the underlying transformation rules are not claimed correct in the literature. A lot of these approaches apply MDE tools (*e.g.* EMF (Steinberg et al., 2008), Acceleo (Eclipse, 2012), ATL (Jouault et al., 2006), etc) to ensure the transformation into B, which do not favour formal verification especially as the latter are not supported by provers and/or model-checkers. Obviously Meeduse can be applied to rethink transformational approaches using the B language so that the correctness of the transformation can be ensured. In my opinion this observation is reasonable considering that mostly these approaches are done by persons who have good skills in B. Furthermore, not only Meeduse has showed its strength in the field of model transformation, but it is itself built on an MDE architecture that can interoperate with the aforementioned tools; and given that it embeds ProB, it therefore includes the required verification features. The challenge of (correctly) embedding B within modelling languages is addressed in the xOWL project (executable OWL[4]), that I have undertaken recently with the LIMICS[5] unit of INSERM[6]. The project is about domain ontologies and health-care applications. Among its objectives, the xOWL project proposes to revisit, via Meeduse, the two kinds of embeddings (Aït-Ameur et al., 2017): shallow embedding and deep embedding. In a shallow embedding, the ontology is directly translated into B without keeping trace of its semantics, and in a deep embedding, the ontology together with its underlying semantic domain are mapped to a formal B model. The contribution of Meeduse is the usage of B in order to rigorously define the transformation rules, prove their correctness and execute them as well to produce the expected formal models.

Assisted animation of xDSLs via machine learning and ProB. Under the MDE umbrella, executable DSLs (xDSLs) are seen as an efficient way for building a model of the system, because of their ability to simulate the system's behaviour, before jumping to its implementation. Most of the tools dedicated to xDSLs provide interactive animation features (also called debugging) allowing domain experts to understand a behaviour, find the origin of errors or simply play with normal execution scenarios. This feature in Meeduse benefits from the various capabilities of ProB, especially animation, model-checking and constraint solving. However, even

[4] OWL: Web Ontology Language.
[5] LIMICS: Laboratoire d'Informatique Médicale et d'Ingénierie des Connaissances en e-Santé.
[6] INSERM: Institut national de la santé et de la recherche médicale

if Meeduse is built on a well-established formal approach, debugging a model remains a cyclic process, during which, after the model is created or refined, the modeler executes it and navigates through its state space to check if the requirements are satisfied, and if not, the steps are repeated. Practically, "satisfying the requirements" means that, starting possibly from any state, for reaching some target states, the sequence of needed actions to be performed on the model is the desired one. Significant drawbacks of such an approach is that in the debugging loop, it is very likely that the state space will change, thus, the same sequence of actions is no longer valid, or the state space could be considerably large, therefore, a manual exploration could be impractical. A brute force inspection of all the possible paths is out of discussion, thus there is a need for an "intelligent" recommendation of which actions to perform for specific states. With this in mind, machine learning techniques appear as a promising approach to assist interactive animation, by identifying the good actions to execute, towards a specific goal defined by the domain expert. This topic has been investigated in a master's project (M2 MoSIG) that I co-supervised with D. Vaufreydaz from the M-PSI[7] team of the LIG lab. After investigating the state of the art, we realized that assisting the interactive animation of xDSLs via AI techniques were not explored at all; and thus this project is a first attempt in this field. Our work resulted in the development of a reinforcement learning approach that is integrated within Meeduse and whose objective is to provide guidance during the animation (and the debugging) of an xDSL.

B.3. Main publications

Akram Idani. The B Method meets MDE: Survey, progress and future. In *16th International Conference on Research Challenges in Information Science (RCIS)*, volume 446 of *LNBIP*. Springer, 2022a. URL https://doi.org/10.1007/978-3-031-05760-1_29.

Akram Idani. A Lightweight Development of Outbreak Prevention Strategies Built on Formal Methods and xDSLs. In *ACM European Symposium on Software Engineering (ESSE)*. ACM, 2021. URL https://doi.org/10.1145/3501774.3501787.

Akram Idani. Formal model-driven executable DSLs: Application to Petri-nets. *International Journal on Innovations in Systems and Software Engineering (ISSE)*, 18(4), 2022b. doi: 10.1007/s11334-021-00408-4.

Akram Idani, Yves Ledru, and German Vega. Alliance of Model Driven Engineering with a Proof-based Formal Approach. *International Journal on Innovations in Systems and Software Engineering (ISSE)*, 16(3):289–307, 2020. doi: 10.1007/s11334-020-00366-3.

Akram Idani. Meeduse: A tool to build and run proved DSLs. In Brijesh Dongol and Elena Troubitsyna, editors, *16th International Conference on Integrated Formal Methods (IFM)*, volume 12546 of *LNCS*, pages 349–367. Springer, 2020a. URL https://doi.org/10.1007/978-3-030-63461-2_19.

[7] M-PSI: Multimodal Perception and sociable Interaction

Akram Idani. Dependability of Model-Driven Executable DSLs, Critical Review and Solutions. In *14th European Conference on Software Architecture, Companion Proceedings*, volume 1269 of *CCIS*, pages 358–373. Springer, 2020b. URL https://doi.org/10.1007/s11334-021-00408-4.

Akram Idani, Germán Vega, and Michael Leuschel. Applying Formal Reasoning to Model Transformation: The Meeduse solution. In *Proceedings of the 12th Transformation Tool Contest, co-located with STAF'2019, Software Technologies: Applications and Foundations*, volume 2550 of *CEUR Workshop Proceedings*, pages 33–44, 2019a.

Akram Idani, Yves Ledru, Abderrahim Ait-Wakrime, Rahma Ben-Ayed, and Simon Collart-Dutilleul. Incremental Development of a Safety Critical System Combining formal Methods and DSMLs − Application to a Railway System. In *24th International Conference on Formal Methods for Industrial Critical Systems (FMICS'2019)*, volume 11687 of *LNCS*, pages 93–109. Springer, 2019b.

Akram Idani, Yves Ledru, Abderrahim Ait-Wakrime, Rahma Ben-Ayed, and Philippe Bon. Towards a Tool-Based Domain Specific Approach for Railway Systems Modeling and Validation. In *Third International Conference onReliability, Safety, and Security of Railway Systems (RSSRail'2019)*, volume 11495 of *LNCS*, pages 23–40. Springer, 2019c.

C. How to read this document?

The objective of a HDR is:

> " *L'Habilitation à Diriger des Recherches sanctionne la reconnaissance du haut niveau scientifique du candidat, du caractère original de sa démarche dans un domaine de la science, de son aptitude à maîtriser une stratégie de recherche dans un domaine scientifique ou technologique suffisamment large et de sa capacité à encadrer de jeunes chercheurs.*" (Arrêté du 23 nov. 1988 - art. 1).

Considering this objective, there are several ways to write a HDR; ranging from detailed presentation to a general overview of the major research contributions. The doctoral school ED-MSTII, advocates for two possibilities[8]:

> " *Le mémoire d'habilitation peut correspondre à un document de type « thèse » (monographie) ou être une compilation d'articles de recherche significatifs accompagné d'une synthèse de quelques dizaines de pages.*"

[8] https://edmstii.univ-grenoble-alpes.fr/MSTII-formulaires/Procedures-HDR-MathsAppli-Info.pdf, Section 3, Page 3.

This document is a mixture of the two possibilities. Some chapters refer to already published papers, and some other chapters present a self-contained technical contribution. In all cases, most of the work presented in this document has been published. The preamble of every chapter provides a list of papers to which the reader may refer in order to get more details about the discussed results. As stated in this introduction, I worked in two fields: MDS and MDE; hence, the remainder is structured in two parts:

- Part I (Formal Modeling of Secure IS) presents my contributions in MDS
- Part II (Building correct DSLs) presents my contributions in MDE and DSLs.

1st part

Formal Modeling of Secure Information Systems

Chapter 1

A Model-Driven Architecture for UML-to-B

Contents

Method integration has been a challenge since several years. The objective is to link formal and graphical paradigms in order to guarantee the quality of specifications. Indeed, on the one hand, graphical languages (such as UML) have been widely used for specifying, visualizing, understanding and documenting software systems, but they suffer from the lack of precise semantical basis. Hence, these languages are more convenient for large-scale software than for industrial applications which address safety challenges. On the other hand, formal methods (such as B (Abrial, 1996)) are specifically used for safety critical systems in order to rigorously check their correctness but they lead to complex models which may be difficult to read and understand. These complementarities between formal and graphical languages motivate a lot of research teams to develop tools which combine both languages. Indeed, disadvantages of semi-formal languages can be avoided thanks to contributions of formal languages and vice versa. In my works I focused my interest on tools dedicated to produce B models from UML diagrams.

The disparity between these two paradigms provides several transformation strategies from UML models to the B language and explains the existence of various UML to B translation

approaches (Laleau and Polack, 2002, Lano et al., 2004, Ledang, 2001, Meyer, 2001, Snook and Butler, 2006b). Each approach proposed its own transformation rules and encoded them in a dedicated tool. Despite the difference between the resulting B specifications, their transformation rules are overlapping and can be mixed and customized for a better coverage of UML concepts. For a more flexible application of model transformation tools, the works of M.-A. Labiadh (supervised by Y. Ledru and myself, and supported by the ANR-SELKIS project) proposed a unifying MDE platform in which one can enhance and customize the existing UML-to-B approaches. This chapter provides the general ideas behind the platform.

Related publications

Akram Idani, Mohamed-Amine Labiadh, and Yves Ledru. Infrastructure dirigée par les modèles pour une intégration adaptable et évolutive de UML et B. *Ingénierie des Systèmes d'Information*, 15(3):87–112, 2010. URL https://doi.org/10.3166/isi.15.3.87-112. Extended version.

Akram Idani, Yves Ledru, and Mohamed-Amine Labiadh. Ingénierie dirigée par les modèles pour une intégration efficace de UML et B. In *Actes du XXVIIème Congrès INFORSID*, pages 261–276, 2009.

Mohamed-Amine Labiadh, Akram Idani, and Yves Ledru. Approche transformationnelle à base de méta-modèles pour l'intégration de UML et de notations formelles. In *Actes de la Conférence AFADL'10: Approches Formelles dans l'Assistance au Développement de Logiciels*, pages 197–212, Poitiers, June 2010.

Structure

This chapter is structured as follows:

- Section 1.1 gives a quick overview about the B method and its underlying tools.
- Section 1.2 is a survey about existing UML-to-B techniques for class diagrams.
- Section 1.3 discusses a model-driven architecture that we developed to gather in a unifying framework, the various approaches.
- Section 1.4 shows how multi-transformations are defined and executed within our platform.
- Section 1.5 draws the conclusions of this chapter.

1.1 Brief overview of the B method

This section gives a brief overview of the B method focusing on the notions that are useful for the readability of this document. The reader can refer to the B-Book (Abrial, 1996) for more details about the B theory. Roughly speaking, B is a formal method based on mathematical foundations (set theory and first order logic) and where the system state is modeled via predefined abstract data types. It covers both static and dynamic aspects of a system. The static aspects are characterized by a set of variables and constants (called *data*) and the dynamic aspects are performed by a set of operations built on the generalized substitution theory.

1.1.1 Abstract machines

The notion of abstract machine is fundamental in B. It provides a structured development composed of three parts: header, declarations and operations. Abrial (1996) compares abstract machines to a pocket calculator with an invisible memory and a number of keys:

> "*The memory (or better the values stored in it) forms the state of this machine, whereas the various keys are the operations that a user is able to activate in order to modify the state.*" Abrial (1996).

Structural features of abstract B machines

MACHINE
- HEADER PART
  - Name of the machine
  - Parameters
  - Constraints
- STATIC PART (DECLARATIVE)
  - Sets
  - Constants
  - Properties of the constants
  - Variables (*state*)
  - Invariant (*properties of the state*)
- DYNAMIC PART (OPERATIONAL)
  - Initialization of the variables
  - Operations

END

1. The header part: introduces the name of the machine, its parameters and the constraints of these parameters;

2. The static (declarative) part: includes sets, constants and variables. It also lists the properties of the constants and a characterization of the machine state using invariants. The latter are expressed in the first order predicate logic;

3. The dynamic (behavioural) part: includes the initialization of the variables and the various operations. The task of an operation is to modify the state of the machine without violating its invariant properties.

1.1.2 Structured developments

The B method masters the complexity of the formal development process using structured specifications. There are two kinds of structures: refinement and composition. Refinement is used to transform an abstract model into a more concrete one, and composition makes possible to construct an abstract machine from smaller, already defined and proved, ones.

1.1.2.1 Refinements

Starting from a high-level abstract specification, and following a refinement process until the production of an executable program, the B method, is a well-established development process. It has the ability to cover all the stages of a software life cycle. Indeed, B formal documents (Figure 1.1) may define several abstraction levels ranging from preliminary design to coding. The transition from one level to another is called refinement, and is guided by proof obligations whose purpose is to establish a correct relationship between the abstraction levels. This decomposition into successive proved refinements allows modular and safe developments.

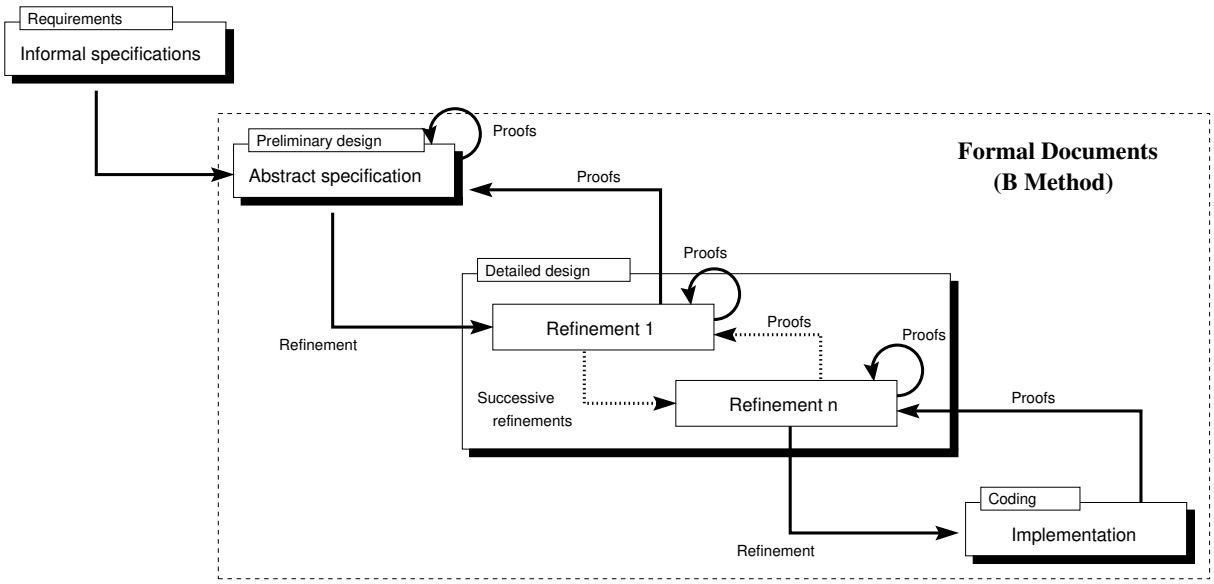

Figure 1.1: Formal development process

In B, there are two kinds of refinements: behaviour refinement and data refinement. The behaviour refinement means that the operation body is changed by an other one but without violating neither the conditions under which the initial operation is defined nor its possible executions. Data refinement applies a new set of data in the refined model, that is some data

can be replaced and some data can be added. When data are being replaced, a linkage invariant must be defined to specify a conformity relationship between the old data and the new ones.

1.1.2.2 Composition

The B method provides an assembly mechanism aiming at the creation of structured and interdependent components. The objective, as stated in (Abrial, 1996), is to compose the specifications of two (or more) abstract machines and thus, eventually, construct abstract machines in an incremental way. The composition mechanism in B, establishes the rules under which an abstract machine can access (read, write) data and operations issued from other machines. There are five clauses that ensure this composition: INCLUDES, SEES, USES, EXTENDS, PROMOTES and IMPORTS.

The mostly used composition construct in the current document is the inclusion (clause INCLUDES). The latter is illustrated in Figure 1.2 where two abstract machines M and N are presented, and such that N includes M. The visibility rules are then as follows: (*i*) the parameters of M are not accessible outside M, (*ii*) the variables of M can be read by N but their modification is only possible via operation call, (*iii*) the properties of N can refer to sets and constants from M, and (*iv*) the invariants of N can refer to sets, constants and also variables from M.

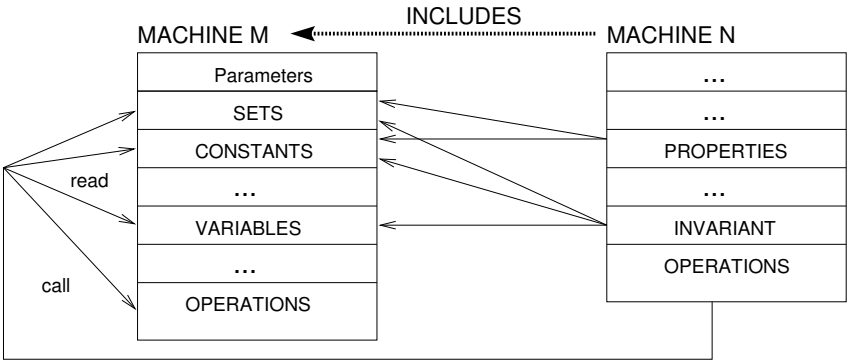

Figure 1.2: Clause INCLUDES.

1.1.3 Generalized substitutions

1.1.3.1 General concepts

A B machine behaves like a state/transition model whose states are represented by the various variables of the machine and the transitions are ensured by its operations. To this aim, B applies the generalized substitution theory, inspired by the Hoare logic (Hoare, 1969) and semantically built on the weakest precondition ($\mathcal{W}p$) calculus of Dijkstra (1975). Tables 1.1 and 1.2 give some major substitutions of the B language. Formula $[S]R$ determines the $\mathcal{W}p$ that ensures the termination of substitution S and such that assertion R is true after the termination of S. For

example, in the definition of the "becomes equal" substitution, $[x := e]R$ denotes the formula obtained after replacing all free occurrences of x in R by expression e. This is defined as $R[e/x]$ (Cf. the B-Book Abrial (1996), § 1.3.3).

Substitution	Formal definition	B notation	Semantics ($\mathcal{W}p$)
Becomes equal	$x := e$	$x := e$	$[x := e]R \Leftrightarrow R[e/x]$
Sequencing	$S;T$	$S;T$	$[S;T]R \Leftrightarrow [S][T]R$
No effect	$skip$	$skip$	$[skip]R \Leftrightarrow R$
Preconditioned	$P\|S$	PRE P THEN S END	$[P\|S]R \Leftrightarrow P \wedge [S]R$
Bounded choice	$T[]S$	CHOICE T OR S END	$[T[]S]R \Leftrightarrow [T]R \wedge [S]R$
Guarded	$P \Longrightarrow S$	SELECT P THEN S END	$[P \Longrightarrow S]R \Leftrightarrow (P \Rightarrow [S]R)$
Unbounded choice	$@z \cdot S$	VAR z IN S END	$[@z \cdot S]R \Leftrightarrow \forall z \cdot [S]R$
Where : x is a variable; e is an expression; S and T are substitutions; P et R are predicates.			

Table 1.1: Primitive substitutions

B Notation	Formal definition
IF P THEN S ELSE T END	$P \Longrightarrow S[]\neg P \Longrightarrow T$
CHOICE S OR T OR ... OR U END	$S \,[]\, T \,[]\, \ldots \,[]\, U$
ANY z WHERE P THEN S END	$@z \cdot (P \Longrightarrow S)$
$x :\in E$	$@z \cdot (z \in E \Longrightarrow x := z)$
ASSERT P THEN S END	$P\|(P \Longrightarrow S)$
Where x a variable; E a set ; S, T and U substitutions; and P a predicate.	

Table 1.2: Other substitutions

1.1.3.2 Correctness

One major advantage of B is theorem proving, which refers to the demonstration of logical formulas (called proof obligations, POs) to ensure a correctness claim for a given property (such as an invariant property). Proof obligations are defined by means of $\mathcal{W}p$ formulas that are generated from the various constructs of the B machine.

POs of the initialisation. Having the predicates I, R and C, which denote respectively the invariants, the constants and the constraints of a machine, and S the substitution of the initialisation; then the latter is proved correct, if and only if formula $R \wedge C \implies [S]I$ holds. This proof claims that starting of any state s where both the properties of the constants and the constraints over the parameters of the machine are true, then the execution of S from s terminates in a state such that the invariant I is true.

POs of the operations. The correctness of an operation guarantees that the invariant remains true after the execution of the operation. The invariant must be also true before. Considering

that the general form of an operation is $r \longleftarrow op\_name(p) =$ PRE P THEN A END (where P is a precondition and A a substitution), then the underlying PO is: $I \wedge R \wedge C \wedge P \implies [A]I$.

1.1.4 Illustration

1.1.4.1 A simple example

To illustrate the notions discussed in the previous subsections, let's consider the simple example of Figure 1.3. This B model is composed of two abstract machines that are related to each other via the inclusion mechanism. The left side of the figure (machine *COUNTER*) defines a simple counter (an integer) with three basic operations: *setValue*, *increase*, and *decrease*. The right side is a high level model (machine *RESERVATION*) of a place reservation system providing two services: *book* and *cancel*.

MACHINE
COUNTER
VARIABLES
count
INVARIANT
count $\in$ INTEGER
INITIALISATION
count := 0
OPERATIONS
setValue(*val*) =
PRE *val* $\in$ INTEGER **THEN**
count := *val*
END ;
increase =
PRE *count* $<$ MAXINT **THEN**
count := *count* + 1
END ;
decrease =
PRE *count* $>$ MININT **THEN**
count := *count* - 1
END
END

MACHINE
RESERVATION(*Max*)
INCLUDES
COUNTER
CONSTRAINTS
Max $\in$ **NAT**
INVARIANT
count $\in 0 \,..\, Max$
INITIALISATION
setValue(*Max*)
OPERATIONS
book =
PRE *count* > 0 **THEN**
decrease
END;
cancel =
PRE *count* $<$ *Max* **THEN**
increase
END
END

Figure 1.3: A simple example.

The invariant of machine *COUNTER* is a simple typing invariant. However, to guarantee its correctness operations *increase* and *decrease* must respect the limits MININT and MAXINT,

which justifies their respective preconditions. The invariant property of machine *RESERVATION* restricts the counter values in the interval $0..Max$, where *Max* is a parameter of the machine. The dynamic part (initialisation and operations) of machine *RESERVATION* applies the basic operations provided by machine *COUNTER*. This is done via the inclusion mechanism, which provides a read access the the variables of the included machine and gives the possibility to call its operations.

1.1.4.2 Verification

The B method is assisted by several verification tools such as provers, animators and model-checkers. Butler et al. (2020) give an overview about these tools, together with their history, evolution and industrial usage. In my works I mainly used AtelierB for theorem proving and ProB (Leuschel and Butler, 2003) for animation and model-checking. Figure 1.4 is a screenshot of the interactive prover of AtelierB showing the POs of machine *RESERVATION*. The "Precondition predicate" refers to the claim that an operation preserves the preconditions under which it can call other operations and the "Invariant is preversed" refers to the $\mathcal{W}p$ calculus as discussed above. From machine *RESERVATION*, AtelierB generated 6 proof obligations and it was able to prove all of them automatically.

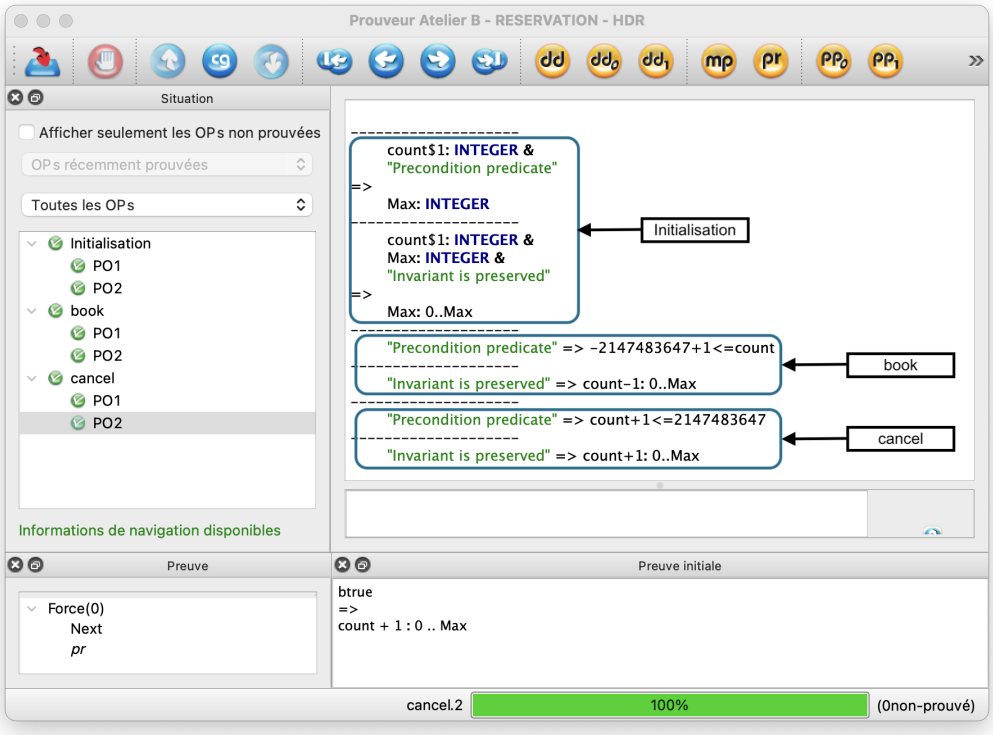

Figure 1.4: Screenshot of the AtelierB interactive prover

Regarding model-checking, its aim is to exhaustively explore of the state space of a B machine, which requires bounded data structures (*e.g.* given sets rather than abstract sets, etc).

Figure 1.5 shows the Java-FX interface of ProB (Leuschel and Butler, 2003) while checking the invariant preservation of machine *RESERVATION*. In this specification, parameter *Max* is set at three. In addition to the numerous verification and visualisation features of ProB, an open-source Java-based API (Körner et al., 2020) of the tool is also available, which offers convenient access to the core features of ProB via external programs.

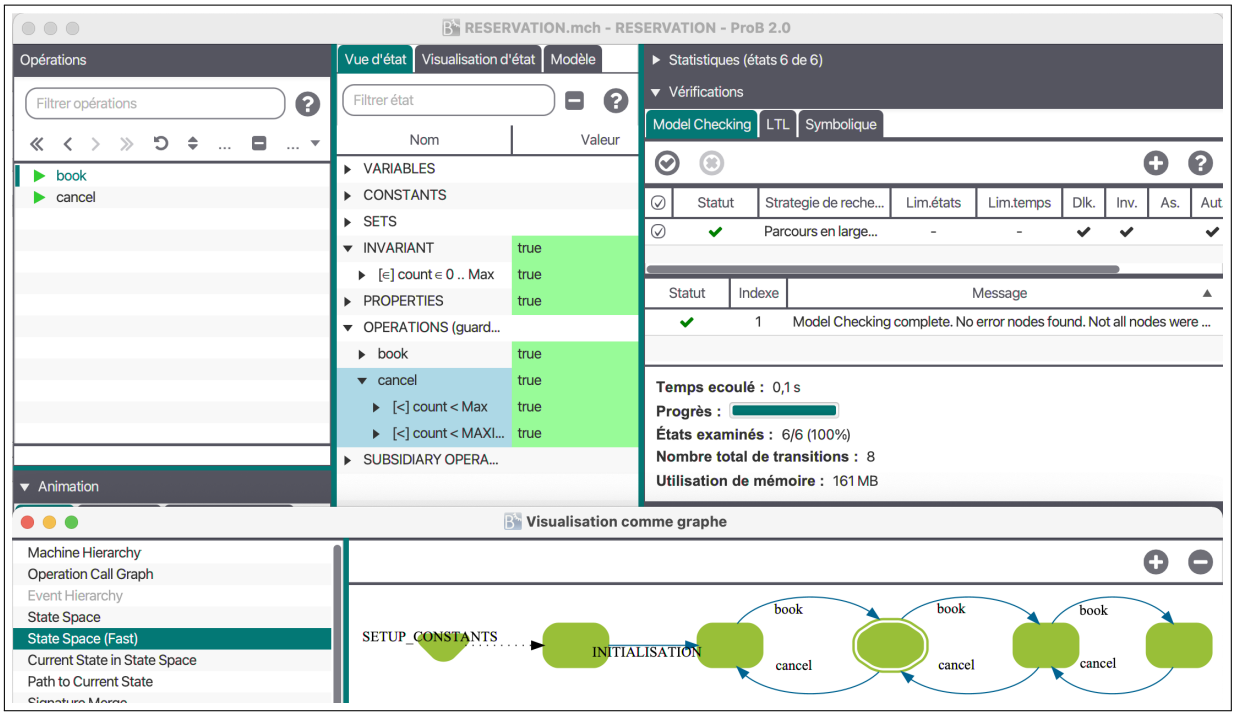

Figure 1.5: Screenshot of ProB

1.2 A quick survey of UML-to-B techniques

1.2.1 Existing approaches

The translation of UML diagrams into B specifications has been addressed since several years. The aim is to strengthen the semantics of UML using a mathematical language and to apply tools of the B method during the verification and validation activities. Several research works have been devoted to this topic and have defined various mappings from UML to B: UML2B (Hazem et al., 2004), UML2SQL (Laleau and Mammar, 2000), U2B (Snook and Butler, 2004) and ArgoUML+B (Meyer, 2001). Every approach has its own objectives and characteristics:

- UML2SQL (Laleau and Mammar, 2000, Laleau and Polack, 2002): this work provides a formal framework for the development of database applications. The B specifications are extracted from UML class diagrams, state-transition diagrams and activity diagrams. Refinement tactics are proposed in order to incrementally generate, from the various B

specifications, a correct implementation of a relational database and a set of safe SQL transactions and queries (such as insertion, deletion, etc).

- U2B (Snook and Butler, 2004, 2006b): this work proposes to produce a B specification, called "natural", so that the proof obligations are as simple as possible. For example, instead of generating a B machine from each UML class (of a class diagram), the authors generate a unique B machine that gathers the B constructs of the whole package. Integrity constraints are expressed in a new formalism (μB) that is built on the B language. Furthermore, C. Snook and his co-authors in (Snook and Butler, 2006b) bring a touch of originality to this technique by specializing UML concepts using stereotypes. These stereotypes provide some guidance to the translation process. In this work both UML class diagrams and state/transition diagrams are addressed.

- ArgoUML+B (Ledang, 2001, Meyer, 2001): this work tried to take into account complex UML features. It proposed, on the one hand, various solutions for the translation of the UML inheritance mechanism, and on the other hand, a new formalization of state/transition diagrams. The starting point is a UML specification (so-called "complete") describing both structural and behavioral aspects of a system. The authors associate to each UML-to-B translation rule a derivation schema showing how the UML constructs are mapped to the B model. Other contributions of this work include the coverage of use-case diagrams and transitions with multiple sequenced actions.

1.2.2 Application

Figure 1.6 applies the three translation tools to a simple UML class with one attribute and shows the resulting B models. One can observe from this figure that there is a consensus about the core B data produced from a UML class, however the relation between these data is translated in different ways. The two first approaches transform a class C into an abstract set A (clauses SETS of figure 1.6) representing the set of possible instances of the class and a variable v (clauses VARIABLES of figure 1.6) representing the set of existing instances of C. The invariant (clauses INVARIANT of figure 1.6) produced by the U2B tool (Snook and Butler, 2004) is $v \in \mathbb{P}(A)$ while the invariant produced by the UML2SQL tool (Laleau and Mammar, 2000) is $v \subseteq A$. Although these two invariants are semantically similar, they are structurally different because they result from two different transformation rules. The ArgoUML+B tool (Meyer, 2001, Ledang, 2001) produces the same invariant as the UML2SQL tool. However, it adds the set of all possible objects *OBJECTS* and considers that A is a constant such that $A \subseteq$ *OBJECTS*. This third translation is structurally and semantically different from the two others.

This example shows that an analyst who is working on the formalization of a UML diagram may be interested by these three transformations depending on the point of view that he/she wants to address. Indeed, one may imagine a fourth transformation in which the U2B and the ArgoUML+B approaches are combined together; meaning that we keep set *OBJECTS* and constant A such that $A \subseteq$ *OBJECTS*, but we produce the invariant of the U2B tool: $v \in \mathbb{P}(A)$.

Similar ideas can be applied to class attributes producing a total function ($\rightarrow$) or a relation ($\leftrightarrow$) depending on the selected approach.

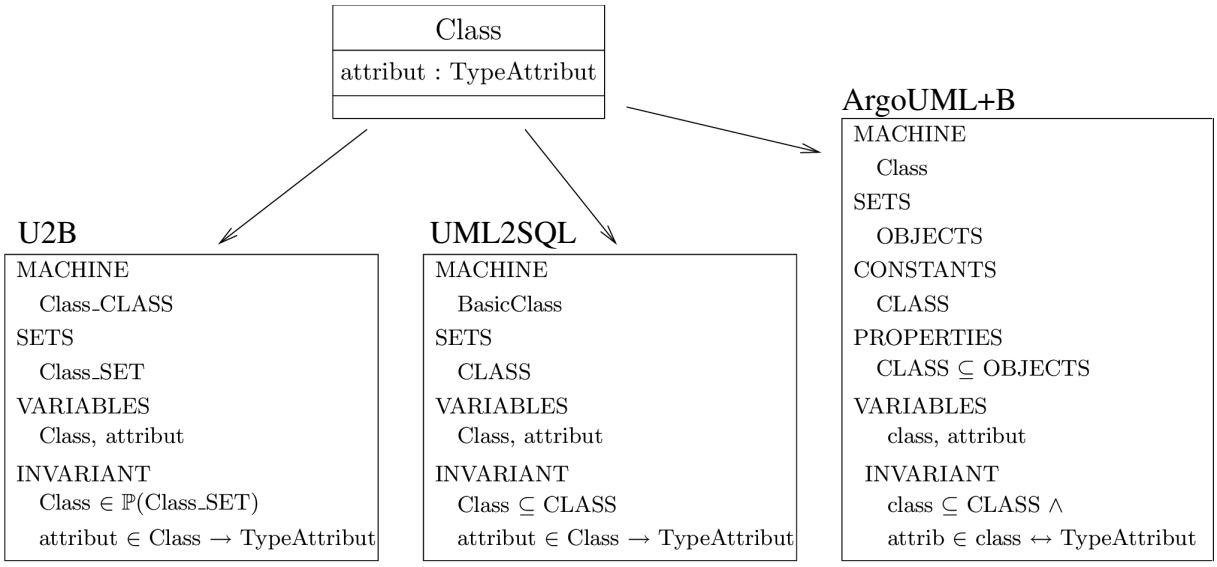

Figure 1.6: Several translations of a UML Class into B

1.2.3 Discussion

Table 1.3 summarizes the structural features of UML class diagrams that are addressed by the three major UML-to-B approaches. The table shows that each approach has its advantages and limitations and that obviously for a better coverage of UML a combination of the various approaches is needed. The research works of M.-A. Labiadh started from this observation and proposed a Model-Driven Architecture (MDA) in which various transformations can be applied and experimented. The MDE paradigm is suitable in this context since MDE widely addressed languages and techniques to define and execute model transformations. In the MDE literature, most works were interested by the developer point view: "*how to encode the transformation rules and apply the underlying execution engines*".

In this work, we are also interested by the user point of view. When several transformations are possible for a given element in the source model, existing tools restrict the transformation to only one choice in order to make the transformation deterministic. However, the end-user may be interested by various possible choices depending on his/her point of view about the good resulting model. This problem is more general than the particular case of UML-to-B. The extraction of Java programs from UML is a good illustration of this claim. One can get different Java programs by applying different tools. Modelio[1] for example provides a unique code generation feature, while the IBM RSA tool[2] provides a configuration feature that allows

[1] Modelio: https://www.modelio.org
[2] RSA Designer: https://www.ibm.com/products/rational-software-architect-designer

the user to personalize the transformation by (un)picking some pre-established options.

	UML2SQL	ArgoUML+B	U2B
Classes (undetermined instances)	+	+	+
Classes (fixed instances)	-	-	+
Class attributes	+	+	+
Distinction between multi/mono-valued attributes	+	-	-
Inheritance	+	+	+
Associations multiplicities	+	+	+
Associations navigation direction	-	+	+
Roles	+	-	+
Associations constraints	+	+	-
Distinction between fixed/non-fixed associations	+	-	-
Association	+	+	-
Associative classes	+	-	-
Parametrized classes	-	-	+

Legend : "+" (considered criterion) ; "-" (non considered criterion)

Table 1.3: Overview of the supported UML structural features

1.3 Towards a unifying method integration approach

1.3.1 Model driven engineering and method integration

The model driven engineering (MDE) approach OMG (2003) defines the software development life-cycle as an iterative process based on model refinement and integration. It makes the distinction between Platform Independent Models (PIM) and Platform Specific Models (PSM).

PIM: "*A platform independent model is a view of a system from the platform independent viewpoint. A PIM exhibits a specified degree of platform independence so as to be suitable for use with a number of different platforms of similar type.*" OMG (2003).

PSM: "*A platform specific model is a view of a system from the platform specific viewpoint. A PSM combines the specifications in the PIM with the details that specify how that system uses a particular type of platform.*" OMG (2003).

A software development process built on MDE is therefore seen as a gradual transformation of a PIM model, which specifies the solution of a system independently of the programming technologies, to a PSM model that describes how this solution can be implemented in a given technology. To this purpose, the MDE approach advocates for the use of rules to define the consistency of models and to gradually transform them during the development process.

Platforms that support this approach require a precise description of the various models using meta-models. Hence, a model transformation rule is a possible projection from a source meta-model to a target meta-model. In the context of method integration (*e.g.* UML and B), the MDE approach has several benefits:

(*i*) Identify the subset of the input model for which the transformations can be applied;

(*ii*) Have a well-defined and executable catalog of transformation rules;

(*iii*) Instrument, via several possible tools, the meta-models and the transformation rules.

1.3.2 The proposed model driven architecture

Existing MDE platforms dedicated to model transformation include a transformation engine that is built on three major elements: (*i*) the model handler such as EMF[3] (Steinberg et al., 2009), (*ii*) the transformation language such as QVT[4] (Gardner et al., 2003), and (*iii*) the language interpreter (or compiler) such as QVTo (Eclipse, 2021). Our MDE architecture for UML-to-B is developed within the EMF platform, where model transformations can be defined at different abstraction levels:

- The lowest level, based on the EMF APIs, often applies the Java programming language;
- The highest level applies a high-level transformation language close to QVT, such as ATL (Jouault et al., 2006) and its virtual machine;
- The intermediate level applies a meta-programming language such as XTend (Efftinge, 2006), which is a dialect of Java that compiles into Java compatible source code.

As our intention is to mix the UML-to-B approaches in one unifying framework, then the underlying transformation rules must fulfil some functional requirements according to the end-user and the rule-writer (the developer) points of view. From a user point of view, the transformation environment must be configurable and offers therefore the possibility to adapt the transformation process by combining rules issued from different approaches. From a rule-writer point of view, the tool must on the one hand offer an extensibility feature to easily introduce new transformations, and on the other hand, provide a way to execute rules that are encoded at different abstraction levels.

Figure 1.7 illustrates the proposed MDE architecture. It is composed of three components: UML, M2M[5] and TMF[6]. Component UML can be any Eclipse based UML editor (*e.g.* TopCased, Papyrus, etc). Component M2M constitues the originality of this work. It includes a transformation engine, so-called "UML/B Transformation Engine", that manages and executes the UML-to-B transformations at different abstraction levels. The inputs of this engine are:

[3] Eclipse Modeling Framework: www.eclipse.org/modeling/emf/
[4] Query/View/Transformation: https://www.omg.org/spec/QVT/1.3/PDF
[5] M2M: Model to model.
[6] TMF: Textual modelling framework.

(*i*) the UML and B meta-models (the latter is roughly discussed in the next subsection),

(*ii*) a catalog of UML-to-B rules that are written in Java (for the lowest level), QVTo (for the highest level) or XTend (for the intermediary level), and

(*ii*) a user-defined configuration file giving the transformation process and the sequence of rules to apply from the catalog (this point is detailed in section 1.4).

Finally, component TMF generates the textual file from a given instance of the B meta-model (called B model in Figure 1.7). It applies a classical model-to-text transformation.

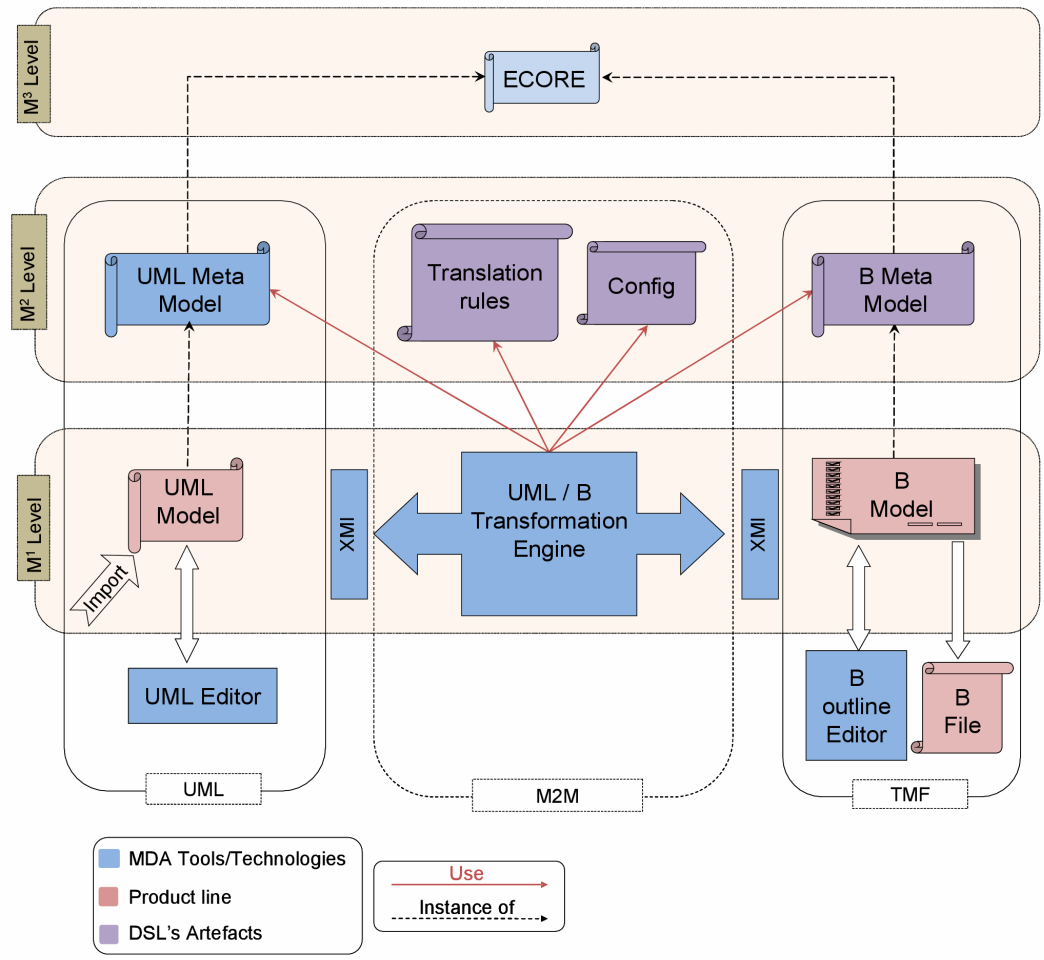

Figure 1.7: Linking UML and B in an MDE architecture

Besides the possibility to reuse and combine rules issued from different UML-to-B approaches, the advantage of this MDE architecture compared with the existing UML-to-B tools (U2B, ArgoUML+B, UML2SQL) is its extensibility. In fact, in order to cover the transformation of UML constructs that have not been considered by the existing approaches, the rule-writer simply adds new rules to the default catalog of transformations that we have implemented; and, if necessary, extends the B meta-model.

1.3.3 A meta-model for the B method

The B meta-model is a core element of our platform. This meta-model has been discussed in (Idani, A., 2006, Idani, A. and Coulette, 2008, Idani, A. et al., 2009, Idani, A., 2009). It is used by components M2M and TMF of Figure 1.7, which are respectively dedicated to the translation from UML into B, and the generation of the B textual files from the resulting B model. This meta-model gives a high-level structural view about the abstract syntax of B. I present an excerpt of it here in order to give an overview of its main concepts and show how the dependencies between the various B constructs are defined.

1.3.3.1 Abstract machines

An abstract machine (Figure 1.8), specified by the meta-class *BMachine*, is composed of static and dynamic parts. The static part contains declarations of sets, constants and variables, and a characterization of these data in terms of constants properties (clause PROPERTIES) and invariants (clause INVARIANT). Meta-class *BData* refers to: abstract sets (*BAbstractSet*), given sets (*BEnumSet*), constants (*BConstant*) and variables (*BVariable*). The dynamic part, is defined by meta-classes *BOperation* and *BInitialisation*, which are used to represent respectively the operations and the initialization of a B machine.

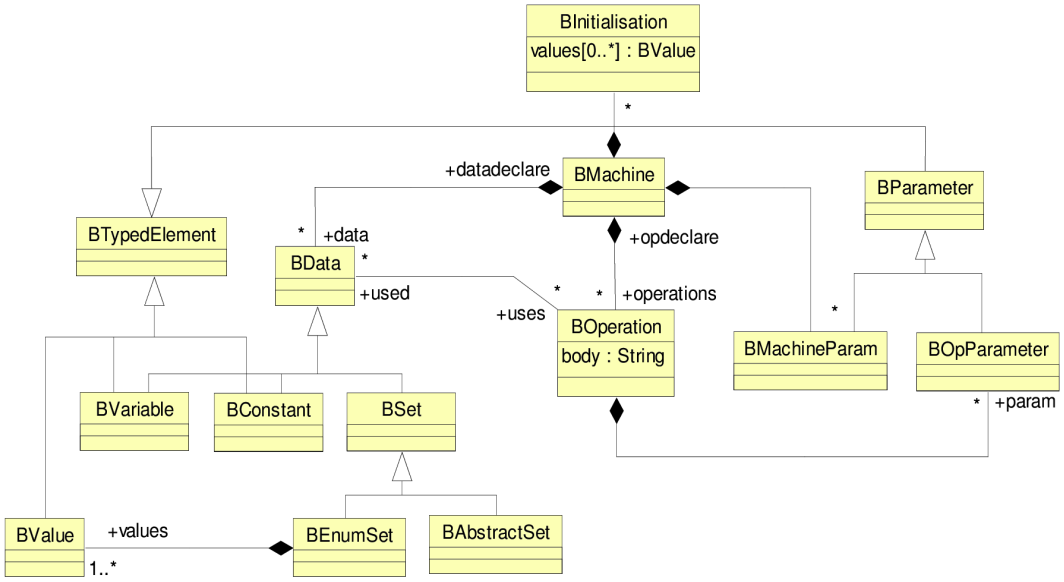

Figure 1.8: B abstract machines meta-model

1.3.3.2 Typing

The B typing meta-model is given in Figure 1.9. It represents the abstract syntax of the most commonly used constructs for typing B data. It does not refer to the B theory. For example, a variable R can be defined in a B machine as:

$$R = \{r | r \in \mathbb{P}(E_1 \times E_2) \wedge$$
$$\forall x, y, z \cdot (x, y \in r \wedge x, z \in r \Rightarrow y = z)\}$$

where E_1 and E_2 are two abstract sets (instances of meta-class *BAbstractSet*). However, such a variable is commonly typed in B with the following invariant:

$$R \in E_1 \nrightarrow E_2$$

and corresponds to a partial function. The B meta-model is then suitable for this second definition of variable R. In fact, it deals with a subset of the B language which is sufficient to cover all the UML-to-B translations addressed by the existing approaches.

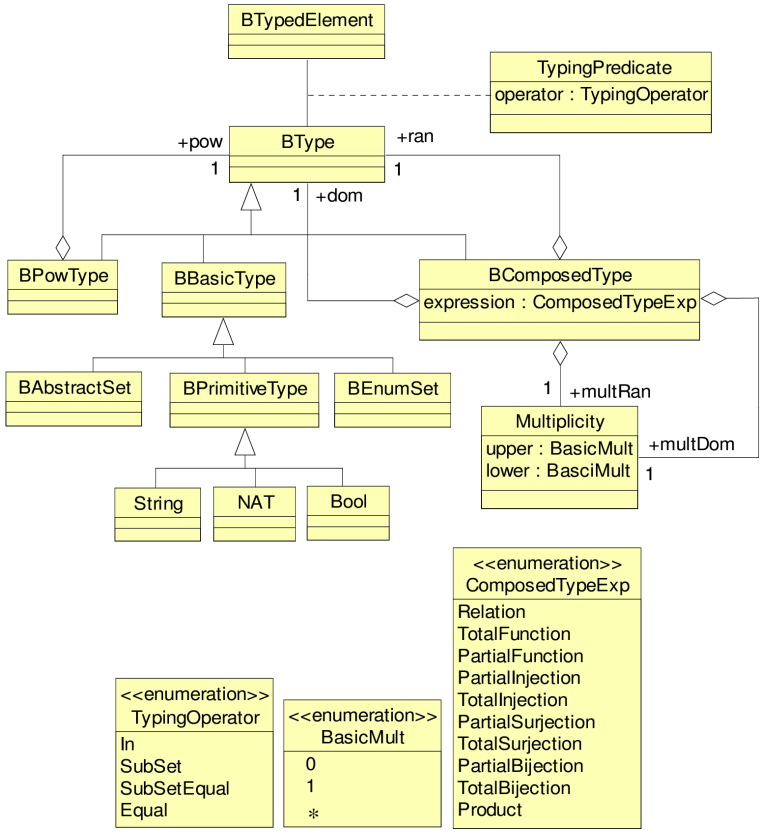

Figure 1.9: B typing meta-model

Typed elements (meta-class *BTypedElement*) are: parameters, constants, variables, predefined B values (e.g. TRUE and FALSE) and values of given sets. Class *TypingPredicate* mainly represents: equality, inclusion and membership. Since the meta-model does not represent the B theory, but the B syntax, it distinguishes between the various typing operators.

Basic types of the B language are represented with class *BBasicType*. They include predefined primitive types (class *BPrimitiveType*) such as BOOL and NAT, and B sets which are mainly abstract sets (class *BAstractSet*) and given sets (class *BEnumSet*). Meta-class *BPowType* represents a type defined using the power-set type constructor $\mathbb{P}$. Some specializations of this meta-class could be defined such as $\mathbb{P}_1$, $\mathbb{F}$ (finite power-set) and $\mathbb{F}_1$.

1.3.3.3 Relations and associations

Class *BComposedType* denotes any type defined from at least two other types. It designates functional relations (*e.e.*"$\leftrightarrow$","$\rightarrow$", etc) and products. Each *BComposedType* is then defined from the types of its domain (role name **+dom**) and its range (role name **+ran**). Regarding Meta-class *Multiplicity*, it describes the cardinalities of the functional relations (Table 1.4). Although these cardinalities are not explicitly defined in B, they are introduced in the meta-model because of their suitability to the translation of UML associations into B relations.

		multDom		multRan	
ComposedType		lower	upper	lower	upper
Relation	$\leftrightarrow$	0	*	0	*
Partial function	$\nrightarrow$	0	*	0	1
Total function	$\rightarrow$	0	*	1	1
Partial injection	$\rightarrowtail$	0	1	0	1
Total injection	$\rightarrowtail$	0	1	1	1
Partial surjection	$\twoheadrightarrow$	1	*	0	1
Total surjection	$\twoheadrightarrow$	1	*	1	1
Partial bijection	$\rightarrowtail\!\!\!\!\twoheadrightarrow$	1	1	0	1
Total bijection	$\rightarrowtail\!\!\!\!\twoheadrightarrow$	1	1	1	1

Table 1.4: Cardinalities associated to specializations of functional relations.

1.4 Multiple transformations

Most of the MDE tools that deal with model transformations apply stand-alone transformations where the transformation is defined as a list (preferably deterministic) of transformation rules. However, often various model transformations need to be composed and integrated during the modeling activities (*e.g.* UML-to-Java). One naive solution for multiple transformations is to encode all the possible transformations and let the user select the one that satisfies his/her objective. However, this is not suitable for us, because on the one hand, this would lead to a multitude of different UML-to-B transformations, and on the other hand, among our requirements we would like to provide an extensibility mechanism so that one can adapt existing rules or extend them. Thus, the transformation could not be stand-alone, and must be configured by the end-user, who decides which rules to apply. Making the transformation configurable, gives a fine-grained application level to its rules. Rather than selecting a pre-established transformation, the user creates his/her own transformation by selecting the rules to apply.

1.4.1 A meta-model for configurable transformations

One major benefit of configurable transformations is that the underlying rules can be reused. In a stand-alone approach common rules must be duplicated, which makes their modification complex and error-prone. However, the difficulty of configurable transformations is to correctly

manage their variations and their uncommon parts. Wagelaar and Straeten (2006) discussed this challenge and presented three techniques that can help manage the composition of model transformations in an MDE context: knowledge-based systems, feature modeling and domain-specific languages (DSLs). These techniques are not directly related to our intention to apply MDE tools in order to deal with multiple transformations. In fact, the underlying solutions are more suitable to model refinements and model refactoring. Nevertheless, they are inspiring as they recommend the definition of a grammar (or a meta-model) that describes the legal configurations of model refinements. This idea is developed and enriched in this work in order to address the multiple transformations of UML into B. Figure 1.10 is a simplified version of our meta-model (Idani, A. et al., 2010a) for configurable transformations. It features two components: `ExecutionContext` and `Configuration`.

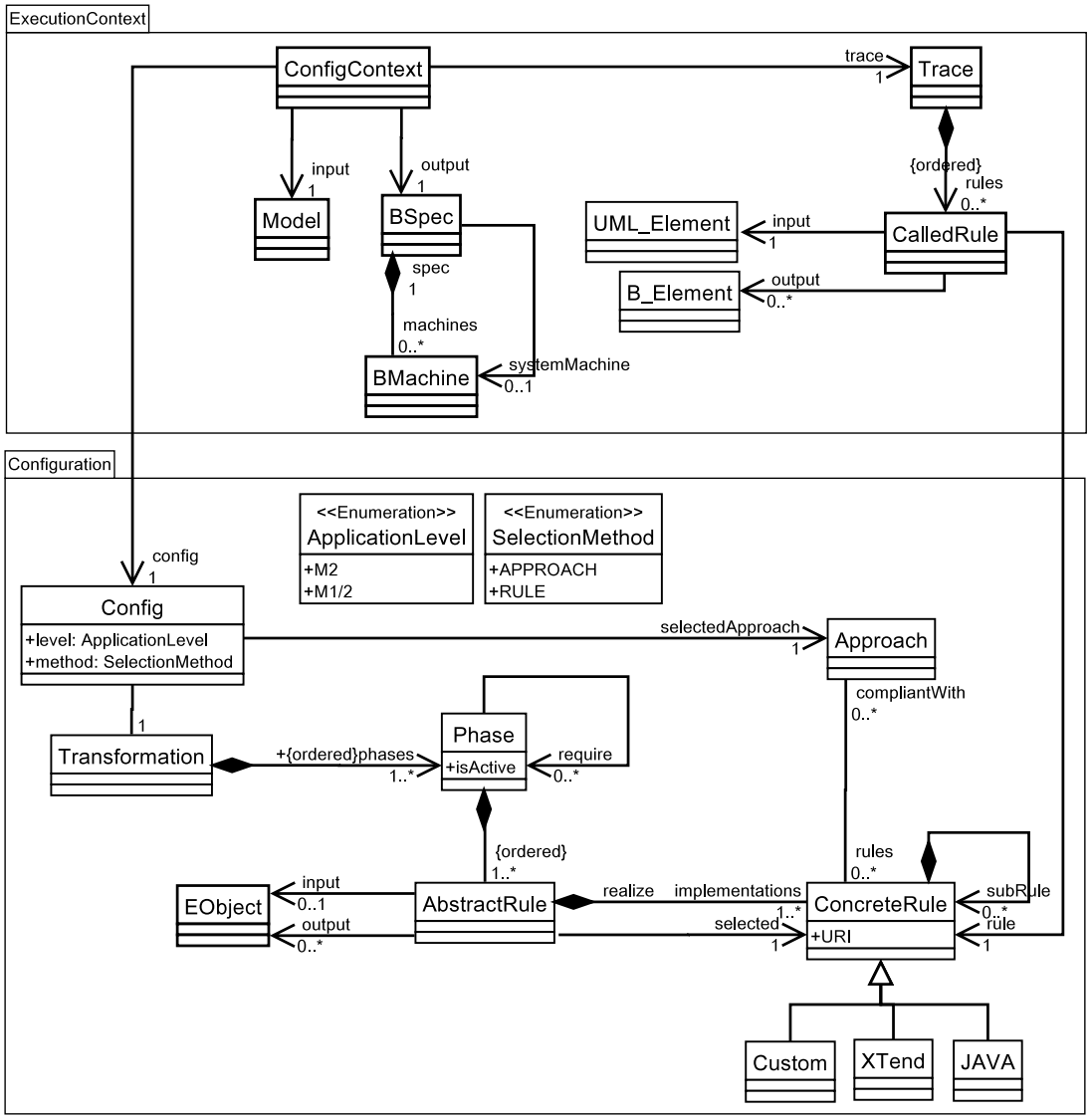

Figure 1.10: Configurable model transformation meta-model

The execution context (whose entry point is class `ConfigContext`), is apart of the API of our transformation engine. It refers to: the input UML model, the output B model (class `BSpec`), the user configuration (class `Config`), and a trace model (class `Trace`) that registers the executed rules together with their inputs and outputs. Regarding the configuration, as stated in Figure 1.7 (Section 1.3.2, Page 30), in addition to the catalog of UML-to-B rules, the user provides a configuration model (called `Config` in Figure 1.7). This model covers three major notions: (*i*) transformation process and phasing, (*ii*) abstraction levels and (*iii*) transformation rules. Note that most of the concepts of this meta-model are generic and can be reused for other kinds of multiple transformations (*e.g.* UML-to-Java). To this purpose only a subset of the execution context need to be specialized.

1.4.2 Transformation process and phasing

In (Czarnecki and Helsen, 2006, Cuadrado and Molina, 2007) phasing is defined as a scheduling mechanism in which the transformation process is divided in several steps, each of which gathers rules that share a specific purpose. Inspired by these works, we define a transformation (meta-class `Transformation`) as a sequence of phases whose order is specified by the "rule writer". Attribute `isActive` of meta-class `Phase` allows one to switch-off a phase and association `require` defines the dependencies that must be checked before running a phase. Figure 1.11 gives a phasing example composed of five steps that depend on the input UML element: packages, classes, associations, attributes and operations.

Top-down arrows represent the ordering of phases, and bottom-up arrows represent the dependencies between them. For example, the transformation of attributes, associations and operations requires the transformation of classes. If the latter is not active then the other phases cannot be executed.

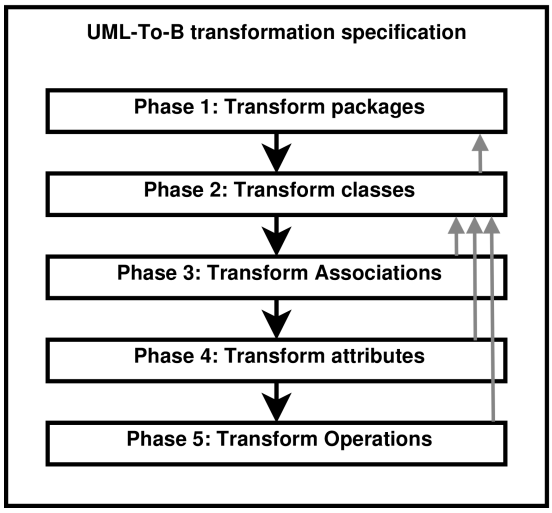

Figure 1.11: A transformation process for a UML class.

1.4.3 Abstraction levels

For a good flexibility of the UML-to-B transformations, we propose three configuration levels (meta-class `Config` and attribute `level`):

- Meta-model level ($\mathcal{M}^2$): means that the transformation rules of a specific configuration are applied uniformly to all elements of the source model. This is the common approach, called MTM (*Model Type Mappings*) in the initial MDA specification (OMG, 2003).
- Model level ($\mathcal{M}^1$): at this level all elements of the source model that are concerned by the configuration are tagged with their corresponding transformation rule. To this purpose our platform embeds a UML profile, which allows one to apply stereotypes to the elements to be transformed.
- Mixed level ($\mathcal{M}^{1/2}$): allows us to customize the $\mathcal{M}^2$ level by forcing some elements from the source model to be translated differently with a specific transformation rule.

The $\mathcal{M}^1$ configuration is hardly applicable to real size models because it needs to tag all the elements of the UML model. It is interesting when a small subset of the source model is translated. The $\mathcal{M}^2$ level allows us to apply existing UML-to-B approaches as they are or to define new approaches. The idea is to encode their corresponding rules and simply execute them on a given UML model. The $\mathcal{M}^{1/2}$ level is, in our opinion, the most interesting one because it provides the possibility to combine rules issued from several transformation approaches. Indeed, one can select a by-default UML-to-B approach (among the ones discussed in Section 1.2) and apply stereotypes to the elements that have to be transformed with different approaches.

Technically, a configuration is an instance of meta-class `Config` of figure 1.10 where attribute `level` can be either `M2` or `M1/2`. The model level ($\mathcal{M}^1$) is applied when all the model elements are stereotyped and the `M1/2` level is selected by the user. Figure 1.12 gives the rule selection process. The choice of the execution level (*identifyExecLevel?*) conditions the transformation according to the two values: `M2` or `M1/2`. For each UML element concerned by the user configuration, the UML/B transformation engine proceeds as follows:

- If attribute `level` is set at `M2`, then the transformation engine applies a transformation strategy (*identifyApproach?*) among two possibilities (attribute `method` of meta-class `Config`): *getRuleByMethod* and *getRuleByAdvice*. The first strategy refers to a well-defined approach, for example the ones discussed in Section 1.2. When this strategy is selected (`method` = `APPROACH`) the transformation engine automatically unfolds all the underlying rules. The second strategy (`method` = `RULE`) is intended to experiment user-defined rules, which are additional rules that do not belong to an existing approach, or a customization of existing ones.
- If attribute `level` is set at `M1/2`, then the transformation engine checks whether the UML element is tagged with a stereotype or not. Operation *getRuleByStereotype* applies

the rule referenced by the stereotype according to a UML profile that we defined to address the $\mathcal{M}^1$ level. This mechanism allows the user to control the transformation stating, element by element, the rule to be applied. However, the main limitation of this mechanism, in the current version of our platform, is that the dependencies between the rules and their compatibilities, must be entirely managed by the user. If the UML element is not stereotyped, one of the *getRuleByMethod* and *getRuleByAdvice* strategies is applied, depending on the user choice.

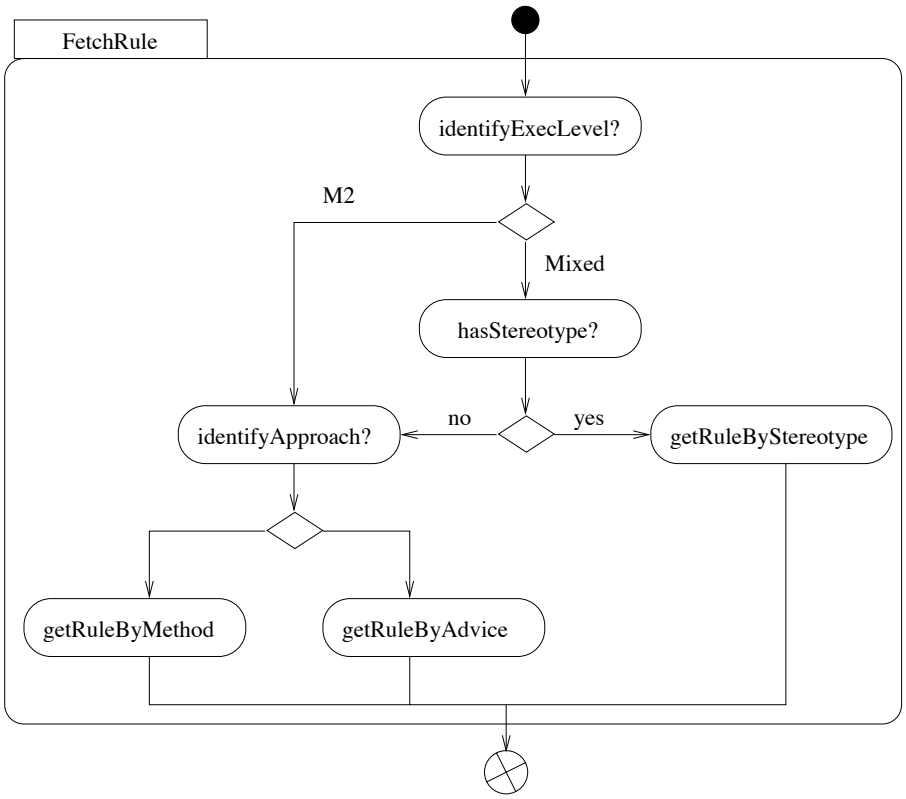

Figure 1.12: Rule selection process.

1.4.4 Transformation rules

A phase is composed of an ordered set of abstract rules (meta-class `AbstractRule`), providing a choice among a set of concrete rules (meta-class `ConcreteRule`). An abstract rule is a mapping from one or several input objects (*e.g.* a UML element) to one or more output objects (*e.g.* B elements). Every abstract rule is implemented by one or many concrete rules, which provides several possible behaviours to a given phase. Attribute `URI` in class `ConcreteRule` refers to the location of the concrete rule. The latter can be encoded in Java, XTend or high-level transformation languages (*e.g.* ATL, QVT); and hence, a transformation can be a mixture of different languages. Our UML/B transformation engine is able to invoke external EMF-based

tools, such as QVTo, as far as a concrete rule respects the signature (inputs and outputs) of the abstract rule that it implements.

In order to illustrate these notions, let's consider two transformation approaches. The first one, inspired by (Snook and Butler, 2004), produces a single B machine for the whole class diagram. We call this approach *UniqueMachine*. It aims to reduce the modularity of the B specification and thus to simplify the theorem proving activity. The second approach, inspired by (Meyer, 2001), produces a B machine from every class, and a root machine that includes the other ones to represent the class diagram. This approach leads to a modular B specification. We call it *MachineByClass*. Both approaches are defined in Figure 1.13 which is an instance of our configurable transformations meta-model. They are represented with the two instances of meta-class `Approach`. In this object diagram a transformation is created (object `MyTransformation`) with two phases, `Phase1` and `Phase2`, which are dedicated respectively to the transformation of packages and classes. In this example we assume that a class diagram is defined in one package. We consider two abstract rules (one per phase): `TransformPackage` and `TransformClass`. The former is realized by a JAVA-based concrete rule (`createSysMachine`), and the latter is realized with two concrete rules that are written in XTend (`ClassToBVariable` and `ClassToMachine`).

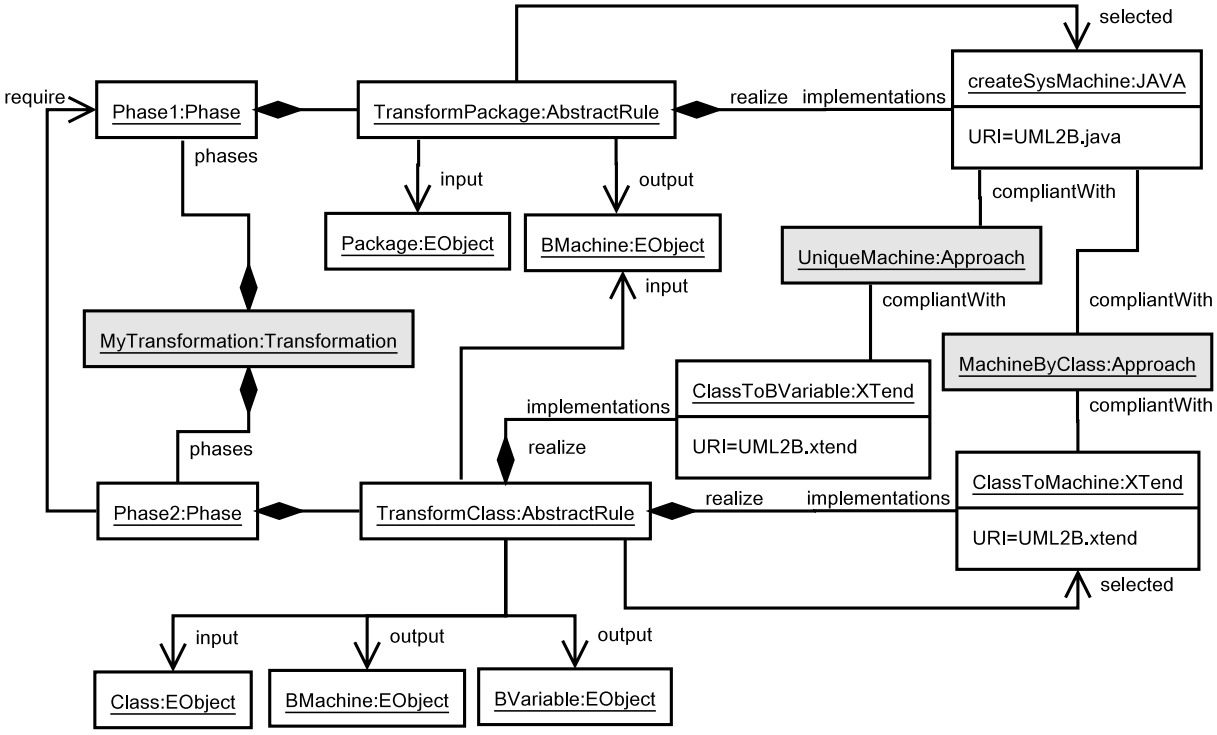

Figure 1.13: An object diagram.

Figure 1.14 shows three configurations applying transformation `MyTransformation`. By running `config1` the rule selection process executes rule `createSysMachine` followed by rule `ClassToBVariable`. Configuration `config2` leads to the execution of

`createSysMachine` followed by `ClassToMachine`. In the first case, when an abstract rule is not realized by any concrete rule that is compliant with the selected approach, the execution process runs the selected default rule (association `selected`). In the second case, the execution process sequentially runs all default rules and does not care about which approaches they are compliant with. For example, `config2` is associated to approach `UniqueMachine`, but it leads to the execution of rule `ClassToMachine`, which is compliant with approach `MachineByClass`. We keep association `selectedApproach` mandatory in order to provide the possibility to quickly switch between the two methods (*i.e.* `APPROACH` and `RULE`). Regarding `config3` it provides the possibility to execute specific rules to particular classes: by default approach `UniqueMachine` is executed but it allows to translate some classes with rule `ClassToMachine`.

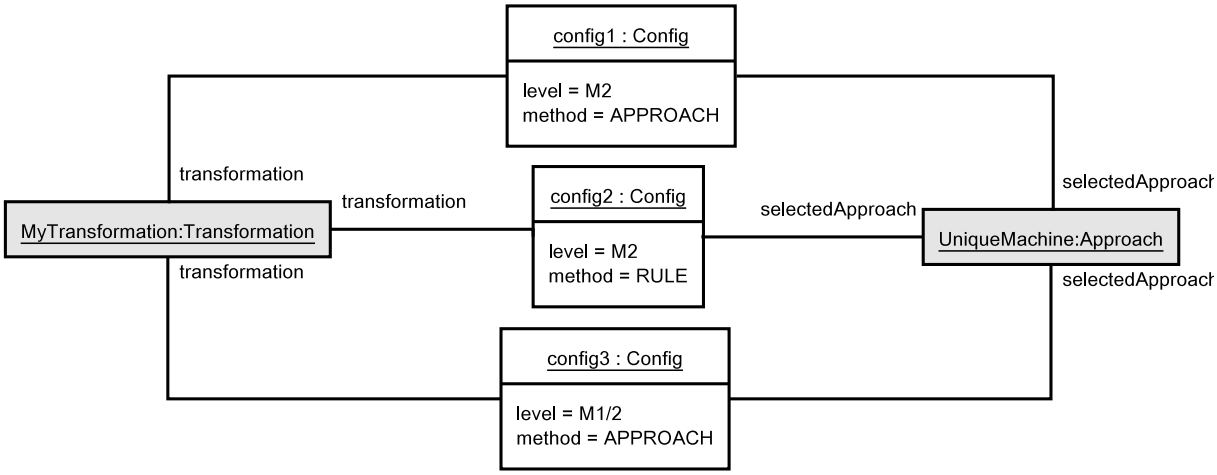

Figure 1.14: Example configurations.

1.4.5 Application

To illustrate the three configurations we consider the class diagram of Figure 1.15 where class `Client` is tagged with stereotype «`ClassToMachine`».

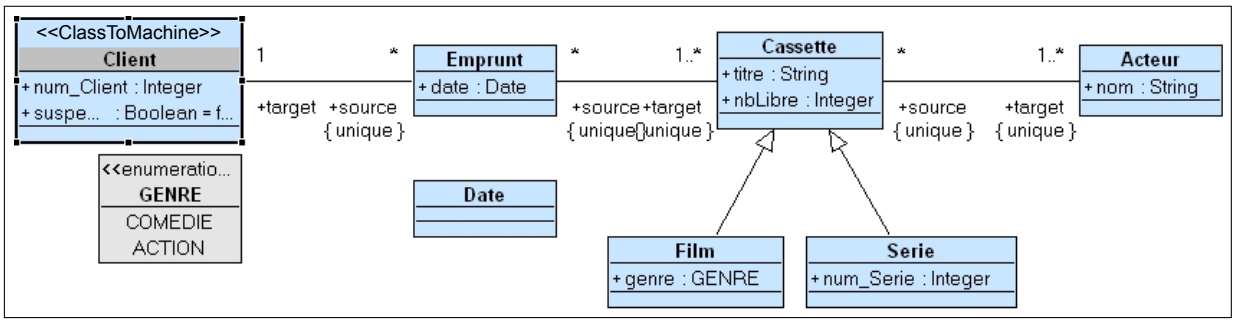

Figure 1.15: A class diagram

The resulting specifications are given in Figure 1.16. Note that the transformation process first extracts an instance of our B meta-model, and then produces the textual files.

config1: Unique Machine

```
MACHINE
        VideoClub
SETS
        CLIENT, EMPRUNT, CASSETTE, ACTEUR, DATE
VARIABLES
         Client, Emprunt, Cassette, Acteur, Film, Serie, Date
INVARIANT
        Client ⊆ CLIENT & Emprunt ⊆ EMPRUNT & Cassette ⊆ CASSETTE & Acteur ⊆ ACTEUR
        & Film ⊆ Cassette & Serie ⊆ Cassette & Date ⊆ DATE
END
```

config2: Machine By Class

```
MACHINE
  VideoClub
INCLUDES
  Client,
  Emprunt,
  Cassette,
  Acteur,
  Film,
  Serie,
  Date
END
```

```
MACHINE
  Cassette
SETS
  CASSETTE
VARIABLES
  Cassette
INVARIANT
  Cassette ⊆ CASSETTE
END
```

```
MACHINE
  Film
USES
  Cassette
VARIABLES
  Film
INVARIANT
  Film ⊆ Cassette
END
```

```
MACHINE
  Client
SETS
  CLIENT
VARIABLES
  Client
INVARIANT
  Client ⊆ CLIENT
 END
```

```
MACHINE Emprunt ...
```

```
MACHINE Acteur ...
```

...

config3: Mixed approach

```
MACHINE
     Client
SETS
     CLIENT
VARIABLES
     Client
INVARIANT
     Client ⊆ CLIENT
END
```

```
MACHINE
        VideoClub
INCLUDES
        Client
SETS
        EMPRUNT, CASSETTE, ACTEUR, DATE
VARIABLES
        Emprunt, Cassette, Acteur, Film, Serie, Date
INVARIANT
        Emprunt ⊆ EMPRUNT & Cassette ⊆ CASSETTE
        & Acteur ⊆ ACTEUR & Film ⊆ Cassette
        & Serie ⊆ Cassette & Date ⊆ DATE
 END
```

Figure 1.16: Three different transformations

1.5 Conclusion

A well-known challenge in software engineering is to have a software specification language that combines precision of formal notations and expressiveness of graphical (or semi-formal)

notations. Several research works have been devoted in order to study possible mappings between these two kinds of notations. This part of my research works has proposed a model driven platform that allows one to integrate several approaches in a single tool. The underlying platform can be used, on the one hand, to combine and extend translation rules from UML diagrams into formal B specifications, and on the other hand, to define transformation rules in several languages (*e.g.* Java, XTend, etc).

More generally, model transformation has received a lot of interest from the research community leading to languages and techniques to encode and execute transformation rules. Most of these works deal with deterministic transformation rules and hence some transformation choices are done implicitly by the rule writer. The originality of the proposed solution is that it deals with multiple transformations and provides a way to the final user to configure the transformation. Multiple transformation means that there are several transformation choices of a source model element. In the existing MDE tools these choices are often done by developers. For example, tools that transform an object model into a relational database may produce a table for every class, however one can be interested by generating tables only from concrete classes and apply other rules to transform abstract classes. Several other examples can be cited such as tools that generate a java source code from UML diagrams: StarUML does not generate getters and setters for private and public attributes while EclipseUML generates automatically getters and setters for private and also public attributes. However, a user may be interested by a mixture of these two transformations: generate getters and setters only for private attributes. The proposal done in this chapter provides a flexible application of model transformation since the choice amongst transformation rules is done by the end-user, not by the developer.

Chapter 2

Formal Model-Driven Security

Contents

Most faults in Information systems (IS) are traced back to deficiencies in specifications (Martin, 2003) and may originate from both functional and non-functional requirements. In a UML-based development process, functional requirements can be defined by means of class diagrams showing the various entities involved in the application logic. Regarding the non-functional requirements my work focuses on security requirements, expressed as access control rules. These rules not only require robust specification methods but also validation and verification techniques to protect systems against malicious attacks. In this context I have been mainly interested by the Role-Based Access Control model (Ferraiolo et al., 2001), which is intended to support access control properties such as confidentiality, integrity, availability and traceability. This topic has been the subject of the Phd thesis of N. Qamar and M.-A. Labiadh that I co-supervised with Y. Ledru and led to the development of the B4MSecure platform (Idani, A. and Ledru, 2015).

In (Qamar et al., 2011a) a comparative study evaluating access-control supported techniques and their validation and verification has been presented and has showed that existing approaches (formal or semi-formal) have significantly dealt with static features. The dynamic features are partially covered and they often do not include functional requirements. In fact, when systems become complex, separation of concerns is often perceived as a good strategy to master complexity. This explains why functional and security models are often validated separately. Although it is definitely useful to first analyse both models in isolation, interactions between

these models must also be taken into account. Indeed, constraints expressed in the security model also refer to information of the functional model. Hence, evolutions of the functional state may influence the security behaviour. Conversely, security constraints can impact the functional behaviour. A typical example is a rule assuming that in order "*to modify a file the current user must be the owner of the file*":

- "*current user*": is a security concern referring to a user that is connected to the system.
- "*modify a file*": is a functional concern dealing with a data, and a possible action.
- "*owner*": is an **authorization constraint** that grants or forbids file modification.

An **authorization constraint** is a constraint that combines functional and security concerns to grant a permission. The proposed FMDE solution to deal with these constraints covers the functional description of the IS as well as its security policy. The supporting models are based on UML for the functional description and SecureUML (Basin et al., 2006, 2009) for the access control rules. A formal B specification is generated automatically from these models, which allows one to formally reason about the whole system: functional and security models can be first validated separately, and then integrated in order to verify their interactions. This chapter describes our solutions and gives the principles of the B4MSecure platform.

Related publications

Yves Ledru, Akram Idani, Jérémy Milhau, Nafees Qamar, Régine Laleau, Jean-Luc Richier, and Mohamed-Amine Labiadh. Validation of IS security policies featuring authorisation constraints. *International Journal of Information System Modeling and Design*, 6(1):24–46, 2015a. URL https://doi.org/10.4018/ijismd.2015010102.

Akram Idani and Yves Ledru. B for Modeling Secure Information Systems, The B4MSecure Platform. In Michael J. Butler, Sylvain Conchon, and Fatiha Zaidi, editors, *17th International Conference on Formal Engineering Methods (ICFEM), Paris, France, November 3-5*, volume 9407 of *LNCS*, pages 312–318. Springer, 2015. URL https://doi.org/10.1007/978-3-319-25423-4_20.

Yves Ledru, Akram Idani, and Jean-Luc Richier. Validation of a security policy by the test of its formal B specification - A case study. In Stefania Gnesi and Nico Plat, editors, *3rd IEEE/ACM FME Workshop on Formal Methods in Software Engineering, FormaliSE'15, Florence, Italy, May 18*, pages 6–12. IEEE Computer Society, 2015b. URL https://doi.org/10.1109/FormaliSE.2015.9.

Jérémy Milhau, Akram Idani, Régine Laleau, Mohamed-Amine Labiadh, Yves Ledru, and Marc Frappier. Combining UML, ASTD and B for the formal specification of an access control filter. *International Journal on Innovations in Systems and Software Engineering (ISSE)*, 7 (4):303–313, 2011. URL https://doi.org/10.1007/s11334-011-0166-z.

Yves Ledru, Akram Idani, Jérémy Milhau, Nafees Qamar, Régine Laleau, Jean-Luc Richier, and Mohamed-Amine Labiadh. Taking into account functional models in the validation of IS security policies. In Camille Salinesi and Oscar Pastor, editors, *CAiSE'11 Workshops, London, UK, June 20-24*, volume 83 of *LNCS*, pages 592–606. Springer, 2011a. URL https://doi.org/10.1007/978-3-642-22056-2_62.

Nafees Qamar, Yves Ledru, and Akram Idani. Validation of security-design models using Z. In Shengchao Qin and Zongyan Qiu, editors, *13th International Conference on Formal Engineering Methods (ICFEM), Durham, UK, October 26-28*, volume 6991 of *LNCS*, pages 259–274. Springer, 2011a. URL https://doi.org/10.1007/978-3-642-24559-6_19.

Nafees Qamar, Yves Ledru, and Akram Idani. Evaluating RBAC supported techniques and their validation and verification. In *Sixth International Conference on Availability, Reliability and Security (ARES), Vienna, Austria, August 22-26*, pages 734–739. IEEE Computer Society, 2011b. URL https://doi.org/10.1109/ARES.2011.112.

Yves Ledru, Nafees Qamar, Akram Idani, Jean-Luc Richier, and Mohamed-Amine Labiadh. Validation of security policies by the animation of Z specifications. In Ruth Breu, Jason Crampton, and Jorge Lobo, editors, *16th ACM Symposium on Access Control Models and Technologies (SACMAT), Innsbruck, Austria, June 15-17*, pages 155–164. ACM, 2011b. URL https://doi.org/10.1145/1998441.1998471.

Yves Ledru, Akram Idani, Rahma Ben Ayed, Abderrahim Ait Wakrime, and Philippe Bon. A separation of concerns approach for the verified modelling of railway signalling rules. In *Third International Conference on Reliability, Safety, and Security of Railway Systems - Modelling, Analysis, Verification, and Certification (RSSRail)*, volume 11495 of *LNCS*, pages 173–190. Springer, 2019. URL https://doi.org/10.1007/978-3-030-18744-6_11.

Abderrahim Ait Wakrime, Rahma Ben Ayed, Simon Collart Dutilleul, Yves Ledru, and Akram Idani. Formalizing railway signaling system ERTMS/ETCS using uml/event-b. In *8th International Conference on Model and Data Engineering - MEDI*, volume 11163 of *LNCS*, pages 321–330. Springer, 2018. URL https://doi.org/10.1007/978-3-030-00856-7_21.

Structure

This chapter is structured as follows:

- Section 2.1 roughly describes the Role-Based Access Control (RBAC) model.

- Section 2.2 presents the principles of the B4MSecure platform.
- Section 2.3 shows how dynamic analyses can be done using B4MSecure and ProB.
- Section 2.4 draws the conclusion and the perspectives of this chapter.

2.1 Role-Based Access Control (RBAC)

2.1.1 Main concepts

RBAC access control mechanism can be used to ensure important properties of data security *i.e.* integrity, confidentiality and availability. Figure 2.1 shows the main concepts of the RBAC model: users (USERS), roles (ROLES), objects (OBS), permissions (PRMS) and operations (OPS). A sixth data type, session (SESSIONS), is used to associate roles temporarily to users, which corresponds to the dynamic part of RBAC. The model differentiates between users and roles: a role is considered as a permanent position in an organization whereas a given user might be switched with another user for that role. Thus, permissions are offered to roles instead of users. A permission refers to operations (application-specific user functions) that can be executed on objects (resources to protect). UA is user assignment, RH is role hierarchy and PA is permission assignment. These concepts are defined in (Ferraiolo et al., 2001) as:

- $UA \subseteq USERS \times ROLES$: a many-to-many mapping between users and roles, UA specifies which roles can be played by a given user;
- $PA \subseteq PRMS \times ROLES$: a many-to-many mapping permission-to-role, PA expresses which roles may be granted a given permission;
- $RH \subseteq ROLES \times ROLES$: a partially ordered role hierarchy, a senior role may inherit the permissions from its junior roles;
- $user\_sessions(u : USERS) \rightarrow Fin(SESSIONS)$: the mapping of user *u* onto a set of sessions, it lists the current session of a given user;
- $session\_roles(s : SESSIONS) \rightarrow Fin(ROLES)$: the mapping of session *s* onto a set of roles, it lists the current roles of a given user in a given session;
- $PRMS = OPS \times OBS$: the set of permissions.

RBAC includes also the principle of separation of duty (SoD), which is intended to enforce conflict of interest policies. There are two main types of separation of duty: SSD (Static Separation of Duties) and DSD (Dynamic Separation of Duties). A SSD forbids a user to be assigned to conflicting roles. It is therefore related to relation UA (and possibly relation RH). A DSD forbids a given user to take conflicting roles simultaneously in the same session.

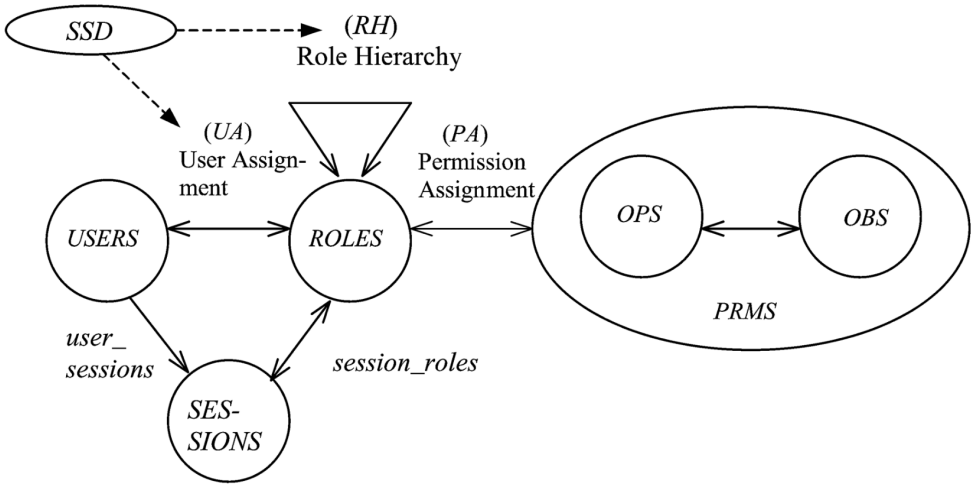

Figure 2.1: Role-based Access Control (Ferraiolo et al., 2001)

Advanced RBAC models allow the specification of constraints such as SoD properties and other properties on roles (*e.g.* precedence). In IS, contextual information may also be taken into account when designing an access control policy. This contextual information may correspond to the current state of the information system, or to the history of interactions with the system. This has led to the notion of **authorisation constraint** in SecureUML (Basin et al., 2006, 2009).

2.1.2 Modeling RBAC with SecureUML

In the literature there exists a number of RBAC representations. SecureUML (Basin et al., 2006, 2009) is one of those techniques that adapt the principles of RBAC and offer a way to model and specify an access control policy. SecureUML provides a security profile extending UML, and applies OCL to define authorization constraints. It appears a good choice to us since it has most of the concepts needed to sketch a security policy using RBAC. Besides, SecureUML is a model-driven approach applying UML diagrams to represent the structural features of an IS, which is coherent with our FMDE approach and the usage of B and UML together for a formal definition of the IS.

By mixing security engineering and model-driven software development, security requirements can be taken into account at a high level of abstraction. Basin et al. (2009) state that "*in this way, it becomes possible to develop security aware applications that are designed with the goal of preventing violations of a security policy*". Further works on SecureUML propose to generate access control infrastructures from SecureUML models and thereby enable a technology independent development of secure systems, which would prevent errors during the realization of access control policies. Figure 2.2 is a UML class diagram representing functional concerns (white classes), to which a SecureUML model is associated for the RBAC rules (grey shaded classes).

This simple model is inspired by a medical IS dealing with patients, doctors, hospitals and

medical records. In this model we suppose that an instance of class *DoctorUser* refers to a user who is assigned to role *Doctor*. Permission *Doctor_Perm* grants a doctor the following actions on a medical record: (*i*) read attributes *data* and *isValid*, (*ii*) modify attribute *data*, and (*iii*) call operation *Validate*. A natural way to deal with authorization constraints in SecureUML is to use OCL. Let us consider, for example, that doctors have the ability to manage only medical records of patients that are admitted in their hospitals. This can be defined by the following OCL expression in the context of permission *Doctor_Perm*.

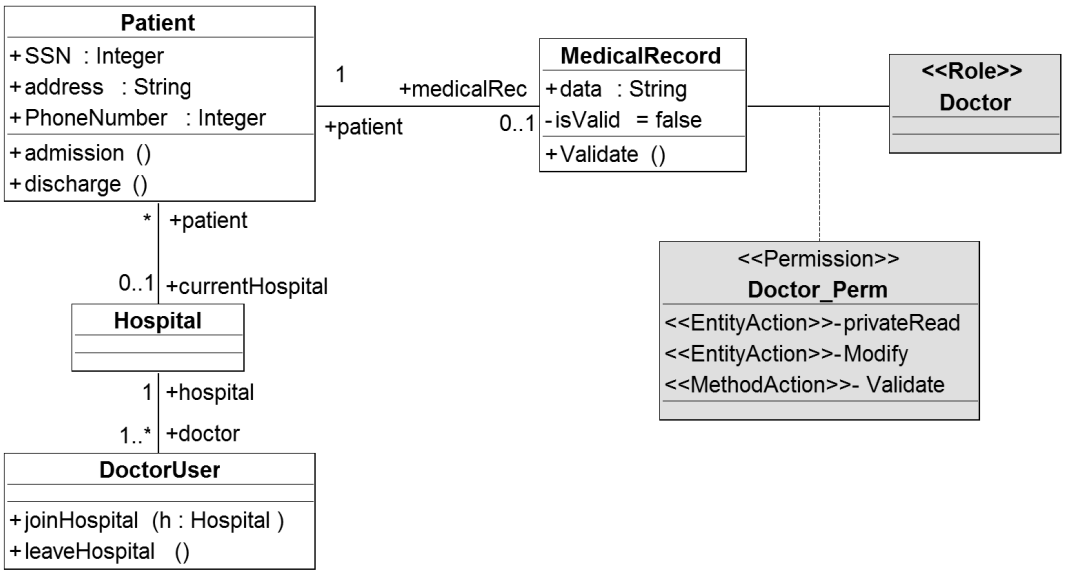

Figure 2.2: Functional model enriched by RBAC rules

```
context Doctor_Perm inv:
session.currentUser.hospital = medicalRecord.patient.currentHospital
```

Expression `session.currentUser` refers to the currently connected user, and as the permission is assigned to role doctor, it can be deduced that the user is an instance of class *DoctorUser*. This constraint navigates through the functional model to retrieve the patient associated to the medical record, and his/her current hospital. It also retrieves the doctor user corresponding to the user asking to access the medical record and retrieves his/her associated hospital. Finally, the constraint compares these two hospitals.

2.1.3 V&V of RBAC policies

Verification and Validation (V&V) of access-control policies have been studied for a long time. Numerous tools exist to ensure these tasks at an implementation level. For example, Sarna-Starosta and Stoller (2004) translate a Security-Enhanced Linux (SELinux) policy into a logic

program and perform queries on this program to verify information flow, integrity properties and separation of duty. The shortcoming of these tools is that they only provide a static analysis focusing on the security policy, without taking into account its relationship with an application or a database. Moreover, they are anchored at the implementation level and hence they are platform-dependent, which is not convenient if the application or the policy must be adapted, corrected or maintained.

2.1.3.1 UMLSec

Jürjens et al. (2005) advocate for an analysis of permissions and security related topics at an early design stage and provide the UMLsec profile (Jürjens, 2010). The corresponding models use UML class and sequence diagrams, decorated with stereotypes and tagged-values to express permissions. The authors verify the conformance between the permissions' model and the system's information flow. In a similar perspective, the Rubacon tool (Höhn and Jürjens, 2008) aims at verifying that systems configurations comply with security rules. The rules and their underlying permissions are defined by a UML class diagram including data extracted from existing applications. Access-control rules are then evaluated by a Prolog analyser on a given instance of the UML model in order to verify permissions, prohibitions and SoD constraints.

Other tools support UMLSec, such as Carisma tool and its predecessor UMLsecTool. However, these tools do not allow to sequence actions or to check whether a given sequence is permitted. Matulevicius and Dumas (2010) performed a comparison of UMLsec and SecureUML and studied possible transformations between them. One of their conclusions is that UMLsec does not cover the notion of authorisation constraint, which is a central concept in our works.

2.1.3.2 OCL-based verification

The Object Constraint Language (Warmer and Kleppe, 1999) is part of UML and allows the specification of invariant constraints on a class diagram as well as pre- and post-conditions of methods. The USE tool (Gogolla et al., 2007) takes as input an object diagram and an OCL constraint. It checks whether the constraint holds on the given object diagram. The tool includes a random generator of object diagrams, and the definition of a sequence of actions creating object diagrams, where pre- and post-conditions can be checked. Sohr et al. (2008) have adapted this tool for the analysis of security policies. Their work is focused on the security aspects, *i.e.* users, roles, sessions and permissions, constrained with OCL assertions, and takes into account functional information by adding some attributes to the concept of users. For example, if a constraint states that the doctor accessing medical information of a patient must be linked to the hospital of the patient, therefore attribute currentHospital should be added to the definition of users. Unfortunately such extensions of the security model do not really scale up, because they duplicate information included in the functional model. In (Sohr et al., 2008) V&V are done based on object diagrams: the diagram is given to the tool, and the tool checks which constraints are violated. The object diagram can be user-defined, randomly generated, or

resulting from a sequence of actions applying a pre-defined action language.

Other works have addressed the validation of security policies using UML and OCL. For example, Ahn and Hu (2007) present an approach using UML class diagrams, a language dedicated to the specification of role-based authorisation constraints (called RCL2000), and OCL to validate SoD constraints. The approach also verifies the constraints given object diagrams. One important limitation of these techniques is that they do not address realistic functional operations. There is no side effect in OCL pre- and post- conditions, which means that the reachability of a given object diagram is not attested. Their V&V start from a given object diagram, but the latter may not be relevant if it is not produced by a valid sequence of functional operations.

2.1.3.3 Applying a formal language

Several formal languages have been adopted to verify the correctness of RBAC variants. A significant amount of works has been carried out using Alloy (Jackson, 2002) and Z (Spivey, 1992). Z has mostly been used to specify RBAC concepts at the meta-level, for example as given in (Yuan et al., 2006); and Alloy has been applied to define and verify security policies at a modeling level. The advantage of Alloy in comparison with other approaches, including semi-formal techniques (*e.g.* OCL, UML), is that Alloy Analyzer is applied to search instances satisfying complex set of predicates (Power et al., 2010). Alloy offers two kinds of automated analysis *i.e.*, simulation and checking. In simulation, operations are evaluated to compute resulting states, and check that they conform to invariant properties. In checking, Alloy attempts to generate instances of a data structure up to a given (small) maximum size, and can identify counterexamples which do not satisfy a given property. The types of answers that Alloy provides are: "*this property always holds for problems up to size X*" or "*this property does not always hold, and here is a counter example*".

Zao et al. (2003) applied Alloy as a constraint analyzer to check inconsistencies among RBAC policies. The authors focus on static properties of the security model and do not take into account evolutions of its state. Schaad and Moffett (2002) address administration tasks of RBAC and arbitrary apply changes to a given model that may result in conflicting situations which may introduce security flaws. Yu et al. (2009) propose scenarios in terms of state transitions to uncover violations in security policies. In this approach, all operation calls take the form of scenarios and a system state is a configuration of objects. Using this technique, one can analyze role activation (and deactivation) and SoD constraints as well. In (Toahchoodee et al., 2009a), functional and security models are merged into a single UML model which is translated into Alloy. Alloy can then be used to find a state which breaks a given property. These properties are mainly of static nature, *i.e.* they focus on the identification of a state that breaks a property, and do not look for sequences of actions leading to such a state. Nevertheless, Alloy can take into account the behaviour of the actions of the model, and we believe it has the potential to perform such dynamic analyses.

2.1.3.4 SecureMova

The tools presented so far, based on UML, OCL and Alloy, only address the validation of security models, *e.g.* addressing SoD properties. Most of them do not consider constraints which involve elements of the functional model, and hence they do not consider evolutions of the state of the functional model. Basin et al. (2009) report on SecureMova, a tool which supports SecureUML. The tool allows a designer to create a functional diagram, *i.e.* a class diagram, and to relate it to permission rules. Constraints can be attached to permissions and may refer to elements of the functional diagram. With SecureMova it is possible to ask questions about a given state, *i.e.* a given object diagram. These questions are called queries; they return authorised actions for a given role, or a given user. They can also investigate on overlapping permissions, *i.e.* permissions which have a common set of associated actions. Unfortunately, all examples discussed in (Basin et al., 2009) are of static nature. In fact, it is not possible to sequence actions (either administrative or functional) or to verify that a given sequence is permitted (or not) by the combination of the security and functional models. In the next sections, we will see that a thorough validation of a security policy which includes contextual constraints must also take into account evolutions of the state of the functional model.

2.1.4 Discussion

Model-Driven Security advocates for the separation of concerns principle in order to ensure modularity and reduce complexity. This is often achieved by isolating functional and security requirements; which allows one to define, verify and implement the various concerns separately. However, in Information Systems, contextual information may also be taken into account when granting permissions. This contextual information may correspond to the current state of the IS, or to the history of interactions with the system. This has led to the notion of authorisation constraint in SecureUML (Basin et al., 2006), combining functional and security concerns together. This notion reveals therefore the drawback of isolating concerns, and shows the need to align functional and security models in order to validate their interactions. There has been a lot of work on the validation of security policies. Some of these works have an associated tool support. In our work, we are mainly concerned with the V&V of security policies involving authorisation constraints. Therefore, we need tools that can take into account both functional and security aspects in their analyses. Moreover, as will be seen in the next sections, evolutions of the functional state of an information system may change the value of an authorisation constraint. It is thus necessary to consider tools which take into account dynamic aspects of the secure information system. Looking at related work, we did not find approaches which combine the dynamic analysis of both functional and security models, with the support for authorisation constraints.

The constraint of Figure 2.2 states that: "*If a doctor wants to modify the medical record of a given patient, he/she must belong to the same hospital as the patient*". Let us now consider a malicious doctor, who wants to modify the information of a patient in another hospital. Since the

patient and the doctor belong to different hospitals, the doctor will not be authorised to access this information. In order to validate the rules of the security policy, one may try several typical situations and query about the permitted/forbidden actions. Using a tool such as SecureMova, one would provide an object diagram od_1 with one doctor and one patient linked to two different hospitals, and query if the doctor may perform action setData on the patient's medical record. The tool would answer that the doctor is not authorised to perform this action. Further validation of this security policy should explore dynamic aspects of the policy. For example, is it possible for this malicious doctor to eventually modify the patient's information? Using only static tools, one can check that, given an object diagram od_2 where the malicious doctor belongs to the same hospital as the patient, he will be granted this access. The next question to investigate is: *does there exist a sequence of actions which leads a malicious doctor to belong to the same hospital as the patient?* This requires to animate a sequence of actions which leads from od_1 to od_2. Such a sequence will presumably call an intermediate operation joinHospital which links the malicious doctor to the hospital of the patient. Here dynamic analyses would allow one to identify these intermediate actions and check which roles have permissions to perform these actions. Another way to group the malicious doctor and the patient in the same hospital is to transfer the patient in the hospital of the doctor. In this second sequence, one should investigate who has the permission to perform such a transfer. This simple example shows that the validation of a security policy may require dynamic analyses to identify sequences of actions leading to an unwanted state. Moreover, these sequences of actions are not restricted to the standard RBAC functions and may refer to operations defined in the functional model. This is actually the case when constraints referring to the functional model are expressed on permissions. Current tools, such as the ones presented in this section, which focus on static queries or on the dynamic execution of the sole RBAC functions are not sufficient to perform such dynamic investigations.

2.2 B4MSecure

2.2.1 Overview

In order to address authorisation constraints in MDS and apply dynamic analysis of the IS, we have developed the B4MSecure platform[1] (Akram Idani and Ledru, 2015). The tool is intended to model the Information System as a whole by covering its functional description, and its security policy. The supporting models are built on UML for the functional concerns and SecureUML for the access control rules. A formal B specification is generated from these models, allowing one to formally reason about the various IS concerns: functional and security models can be first validated separately, and then integrated in order to verify their interactions.

B4MSecure (Figure 2.3) is built on an MDE architecture in which the input models are UML class diagrams that are extended with the SecureUML profile. The extraction of B spec-

[1] B4MSecure: B for Modeling Secure Information Systems

ifications applies a catalog of transformation rules that are defined at a meta-level. The ideas behind the tool are inspired by existing software products, such as popular commercial database management systems (*e.g.* Oracle, Sybase) or webservers (*e.g.* JBoss, Tomcat). The available implementations of RBAC act like a filter which intercepts a user request to a resource in order to permit or deny the access to associated functional actions (*e.g.* transactions on databases, file operations, etc). The tool is based on the same principles, but at a modeling level. Each functional operation is encapsulated in a secure operation checking that the current user has the required authorizations. The usage of B is motivated by the fact that several tools have been built to translate UML models into B specifications as discussed in Chapter 1. Indeed, the translation of the functional model can be done by executing a configuration of UML-to-B rules that are issued from our approach for multiple transformations. Regarding security models, Sohr et al. (2008) have already proved that it can be specified in UML+OCL. Since the B language is based on the same principles as OCL (first order predicate logic and set theory), it is possible to propose a similar translation of the security model into B specifications.

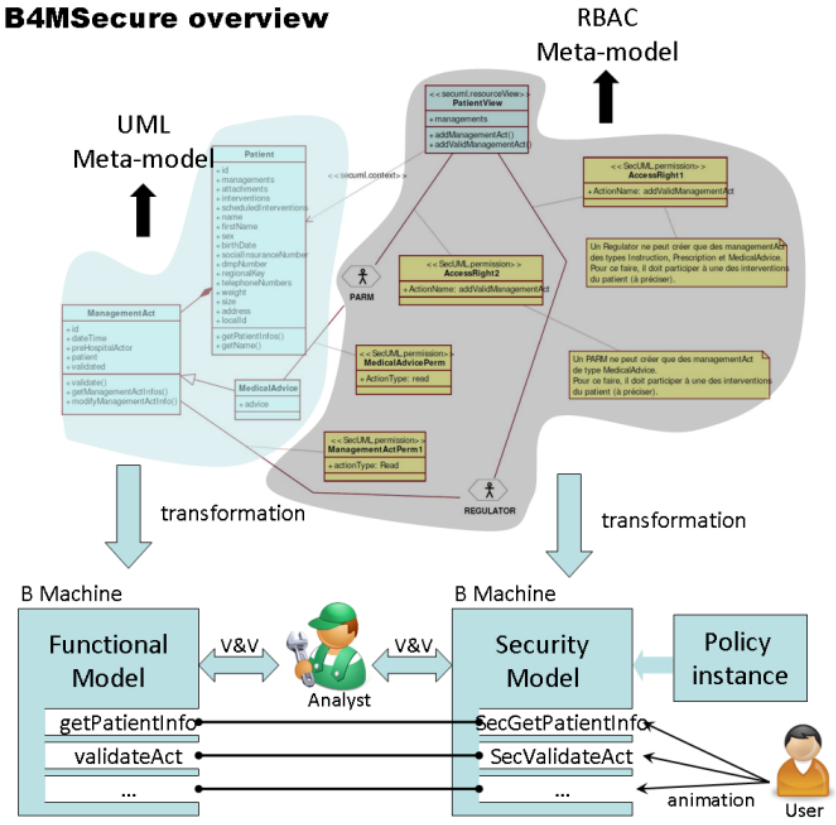

Figure 2.3: Formal V&V activities of functional and security models

The B specifications produced from both security and functional models can therefore be analyzed using either animation tools such as ProB (Leuschel and Butler, 2003) or proof tools such as Atelier B. ProB also includes model-checking facilities which can be of interest to search for malicious sequences of operations. The functional B specification in the left side of

Figure 2.3 applies a classical UML-to-B translation. The lower side of Figure 2.3 represents the formal specification produced from the security model and which is intended to control the functional B operations. Playing scenarios is done by animating the secured operations, which gives only access to the authorized functional operations. This approach allows the validation of the functional model as well as the security policy. In fact, animation of an authorized operation modifies the state of the functional model and hence allows the analyst to validate both models.

Terms "*shallow embedding*" and "*deep embedding*" (Wildmoser and Nipkow, 2004) are often used to describe mappings between formalisms. The first notion means a direct translation from a source model into a target model, while the second notion means that the mapping leads to structures that represent data types. Most UML-to-B approaches adopted the "*shallow embedding*" approach. Indeed, they apply a set of rules to ensure a direct translation from UML into B. Unlike "*deep embedding*", this aims to be more straightforward because the resulting formal specifications explicitly include the elements of the source model. The "*deep embedding*" approach is commonly used to translate elements of a meta-model. To summarize, B4MSecure produces two B models:

1. A first B model issued from a UML class diagram, applying a "*shallow embedding*" approach. This model can be enriched with invariants and operations and the correctness of the functional model can be established.

2. A second B model that represents the security policy and defines the access to the functional entities. This model is generated through a "*deep embedding*" approach. We translate an access control meta-model built on the principles of RBAC, and then we inject in it the B specification of a specific access control model.

2.2.2 Functional model

To illustrate the B4MSecure approach, we consider the UML class diagram of Figure 2.4, inspired by (Bandara et al., 2010). This model represents functional concerns of a banking IS: it defines customers (class *Customer*) in relation with their accounts (class *Account*).

An account is characterized by its balance (attribute *balance*), the authorized overdraft (attribute *overdraft*) and a unique identifier (attribute *IBAN*). A customer may have a credit card (class *CreditCard*), allowing him/her to withdraw cash. Operation *transferFunds* allows one to transfer an amount of money (parameter *m*) from the current account to any account defined with an *IBAN* number (parameter *NB*). Operations *withdrawCash* and *depositFunds* allow respectively to withdraw or to deposit money. To withdraw an amount of money, the customer must present an active credit card (attribute *inLine* must be equal to *true*).

2.2.2.1 Translating the structural features

In this subsection, we are not going to discuss existing UML-to-B approaches. For more details about these approaches, we refer the reader to the previous chapter or to Idani, A. et al. (2010b).

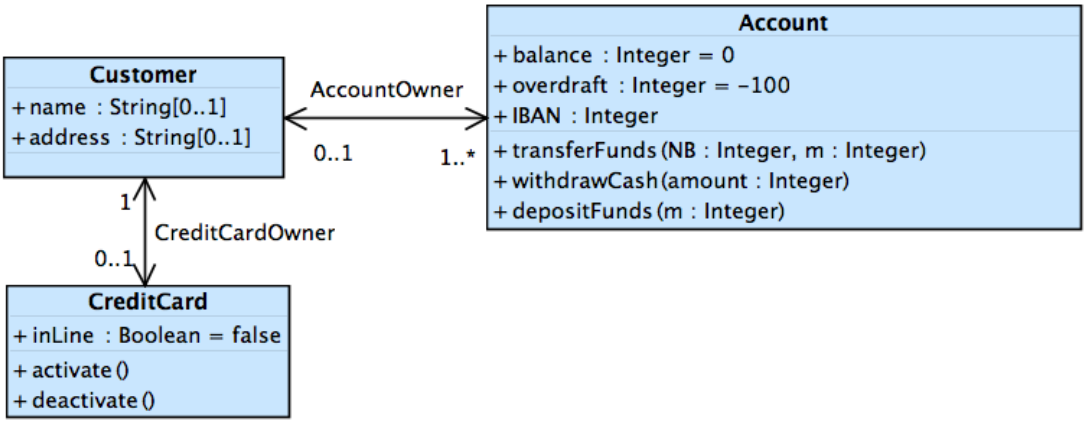

Figure 2.4: UML Class Diagram – Functional Model

Nevertheless, we will focus on the translation set up in B4MSecure, in order to give an overview of the B specifications produced by the tool.

In B, abstract sets represent an abstraction of a set of objects from the real world. As this definition is close to the notion of class in UML, it is used by all UML-to-B approaches to formalize UML classes. The main objective of this translation is to allow the definition of constructors and destructors of class instances. For example, class *Customer* produces:

(*i*) Abstract set[2] *CUSTOMER* defining the set of possible instances;

(*ii*) Variable[3] *Customer* defining existing instances; and

(*iii*) Invariant[4] : $Customer \subseteq CUSTOMER$.

Regarding class attributes, they are translated into B functions relating the set of existing instances to the type of the attribute. The resulting functions depend on the attribute character: mandatory or optional, unique or not unique, single or multiple valuated. For example, attribute *IBAN* of class *Account* is single-valued, mandatory and unique. It is therefore translated into a total injection. Table 2.1 gives the various translations of single-valued attributes.

	Optional	Mandatory
Unique	$\rightarrowtail\!\!\!\!\!+$	$\rightarrowtail$
Not unique	$\nrightarrow$	$\rightarrow$

Table 2.1: B relations extracted from single-valued attributes

The translation of associations follows the same principle. Indeed, each association leads to a functional relation that depends on the multiplicities of the two ends of the association. For

[2] clause SETS.
[3] clause VARIABLES.
[4] clause INVARIANT.

example, association *AccountOwner* is translated into a partial surjective function since its multiplicities are `0..1` and `1..*`. Figure 2.5 presents the typing invariants that are automatically produced by B4MSecure from the example of Figure 2.4.

INVARIANT
$Account \subseteq ACCOUNT$
$\wedge\ Customer \subseteq CUSTOMER$
$\wedge\ CreditCard \subseteq CREDITCARD$
$\wedge\ AccountOwner \in Account \twoheadrightarrow Customer$
$\wedge\ CreditCardOwner \in CreditCard \rightarrowtail Customer$
$\wedge\ Account\_\_balance \in Account \rightarrow \mathbb{Z}$
$\wedge\ Account\_\_overdraft \in Account \rightarrow \mathbb{Z}$
$\wedge\ Customer\_\_name \in Customer \nrightarrow STRING$
$\wedge\ Customer\_\_address \in Customer \nrightarrow STRING$
$\wedge\ CreditCard\_\_inLine \in CreditCard \rightarrow BOOL$
$\wedge\ Account\_\_IBAN \in Account \rightarrowtail \mathbb{N}$

Figure 2.5: Structural invariants produced by B4MSecure

2.2.2.2 Extraction of basic operations

The B specifications produced by B4MSecure from a given class diagram are intended to be animated using an animation tool such as ProB (Leuschel and Butler, 2003). This allows one to see the evolution of the IS and observe the impact that an execution scenario could have on the functional state. Thus, B4MSecure generates all basic operations such as creation/deletion of class instances, creation/deletion of links between these instances, getters/setters of attributes, and getters/setters of links. In general, these operations are correct by construction, meaning that they do not violate the generated typing invariants. In fact, the proof of correctness of the functional model means that basic operations preserve the multiplicities of the associations as well as the character of attributes. This proof is true by construction for most basic operations, but needs to be consolidated for some deletion operations.

For example, deleting an instance of class *Customer* is only possible if this instance is not linked to an instance of class *CreditCard*. Several special cases exist, and are not all covered by the translation process in B. In this case the specifications must be updated or completed by the analyst. Figure 2.6 gives an example of a basic operation that is generated by B4MSecure. It is a creation operation of class *Account*. This operation preserves, on the one hand, the mandatory character of attribute *IBAN* because a value is assigned to the attribute when the object is created, and on the other hand, the uniqueness of this attribute thanks to pre-condition:

$$Account\_\_IBANValue \notin \mathbf{ran}(Account\_\_IBAN)$$

The operation also takes into account the default values of attributes *balance* and *overdraft*; they are respectively initialized to 0 and -100.

Account_NEW(*Instance*, *Account__IBANValue*) ==
PRE
Instance ∈ *ACCOUNT* ∧ *Instance* ∉ *Account*
∧ *Account__IBANValue* ∈ ℕ
∧ *Account__IBANValue* ∉ **ran**(*Account__IBAN*)
THEN
Account := *Account* ∪ {*Instance*}
|| *Account__balance* := *Account__balance* ∪ {(*Instance* ↦ 0)}
|| *Account__overdraft* := *Account__overdraft* ∪ {(*Instance* ↦ -100)}
|| *Account__IBAN* := *Account__IBAN* ∪ {(*Instance* ↦ *Account__IBANValue*)}
END;

Figure 2.6: Basic creator generated by B4MSecure

2.2.2.3 Enhancing the functional model

Information Systems often include constraints, especially integrity constraints relating to the functional model. These constraints must be taken into account when defining the various use cases of the IS. As the verification activities are specific to each IS, the B model produced automatically must be completed manually by adding new invariants and operations. Let's consider for example, that the balance of an account must be greater than (or equal) to the overdraft of the account. The analyst should therefore add the following invariant to the functional B machine:

$$\forall\, aa \,.\, (aa \in Account \Rightarrow Account\_\_balance(aa) \geq Account\_\_overdraft(aa))$$

Figure 2.7: Invariant constraints of the functional model

After adding this invariant to the specifications, the analyst mush check the proof obligations (POs) generated by AtelierB and correct the underlying B specifications. This would identify the basic operations that violate the user defined invariants and correct them accordingly. Consider for example, the modification operation of attribute *balance* that is presented in Figure 2.8. In order to correct this operation, the analyst must strengthen its precondition by adding the following predicate:

$$Account\_\_balanceValue \geq Account\_\_overdraft$$

The enhancement of the functional model is done by adding invariant properties and the underlying preconditions in order to keep correct the basic operations. The user-defined operations, such as operations *transferFunds* and *withdrawCash* of class *Account*, must also be defined. Figure 2.9 presents the B specification of operation *transferFunds*. It takes an account number (parameter *N*) and a positive amount (parameter *m*) and performs the transfer of funds if the following conditions are met: the current account and the beneficiary account are held

by customers, N corresponds to an existing account other than the current account, and the authorized overdraft will not be exceeded by this transfer.

The B specifications issued from our functional class diagram are about 300 lines of B code from which 80 POs were generated by AtelierB for typing invariants. Among these POs, 4 were proved interactively. Furthermore, adding invariant properties produced a total of 94 proof obligations of which only 18 did not pass the automatic prover. Therefore we introduced preconditions to some basic operations in order to have a formal model preserving all its invariants. This formal verification effort may vary depending on the complexity of the IS; for this simple example the verification task was reasonably easy to do.

Account__SetBalance(*Instance*, *Account__balanceValue*) ==
  PRE
    $Instance \in Account \wedge Account\_\_balanceValue \in \mathbb{Z}$
    $\wedge \quad (Instance \mapsto Account\_\_balanceValue) \notin Account\_\_balance$
  THEN
    $Account\_balance(Instance) := Account\_\_balanceValue$
  END;

Figure 2.8: Basic setter of class *Account*

Account__transferFunds(*Instance*, *N*, *m*) ==
PRE
  $Instance \in Account \wedge N \in \mathbb{N} \wedge m \in \mathbb{N}_1$
  $\wedge \quad AccountOwner[\{Instance\}] \neq \emptyset$
  $\wedge \quad N \in \mathbf{ran}(\{Instance\} \lhd Account\_\_IBAN)$
  $\wedge \quad AccountOwner[\{Account\_\_IBAN^{-1}(N)\}] \neq \emptyset$
  $\wedge \quad Account\_\_balance(Instance) - m \geq Account\_\_overdraft(Instance)$
THEN
  $Account\_\_balance :=$
    $\{(Instance \mapsto (Account\_\_balance(Instance) - m))\}$
    $\cup \{(Account\_\_IBAN^{-1}(N) \mapsto (Account\_\_balance(Account\_\_IBAN^{-1}(N)) + m))\}$
    $\cup (\{Instance, Account\_\_IBAN^{-1}(N)\} \lhd Account\_\_balance)$
END ;

Figure 2.9: Operation transferFunds of class *Account*

2.2.3 Security model

The approach adopted in B4MSecure is a model-driven approach that favours the separation of concerns principle. The objective is to first allow reasoning about the correctness of the

functional model without any security concern, and afterwards, to analyze the security model independently of the functional concerns, and finally to check the connections between both models.

UML and SecureUML models can be designed using the graphical editor of TopCased[5] or that of Eclipse Papyrus[6]. Thus, the security policy concrete syntax uses classes and stereotypes to refer to RBAC concepts. Figure 2.10 is a SecureUML model associated to the class diagram of Figure 2.4.

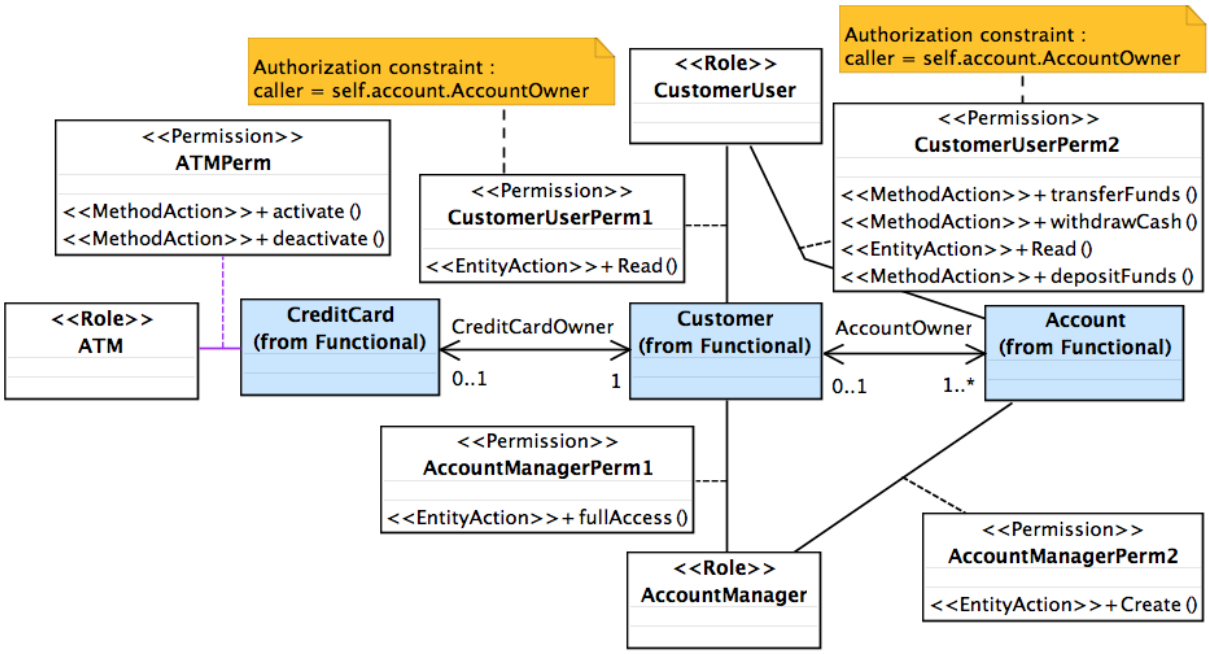

Figure 2.10: SecureUML model in B4MSecure

This model defines three roles: *CustomerUser*, *AccountManager* and *ATM*. They respectively represent the customer of the system, the financial manager in charge of the bank's customers, and the automatic teller machine. The underlying access control rules are:

1. Customers can read their personal data (permission *CustomerUserPerm1*), transfer money, deposit and withdraw cash (permission *CustomerUserPerm2*);

2. The account manager, in charge of the clients, has a full access (read and write) on class *Customer* (permission *AccountManagerPerm1*). He/She can thus create customers, read or modify their data. However, his/her rights on class *Account* are limited to the creation of new accounts (permission *AccountManagerPerm2*);

[5] http://www.topcased.org
[6] https://www.eclipse.org/papyrus/

3. Role ATM has the permission to call activation and deactivation operations of class *CreditCard* (permission *ATMPerm*).

An authorization constraint is associated to permissions *CustomerUserPerm1* and *CustomerUserPerm2* in order to grant the corresponding actions to the sole holder of the account on which they are invoked. In this security policy, the account manager has no access, neither read nor write, to the attributes of class *Account*. We also consider three users and a DSD constraint as presented in Figure 2.11. *Bob* is assigned to role *AccountManager*, and *Paul* and *Martin* are possible users without any assigned role. Note that the assignment of *Paul* and *Martin* to role *CustomerUser* is done when these two users are created as instances of the functional class *Customer*. In fact, in some cases functional classes refer to roles and vice-versa. Hence, we consider that set *CUSTOMER* of possible instances is a subset of set *USERS* that represents possible users of the system. Finally, the DSD constraint of Figure 2.11 means that a user cannot log in the system by being both *AccountManager* and *CustomerUser* in the same session.

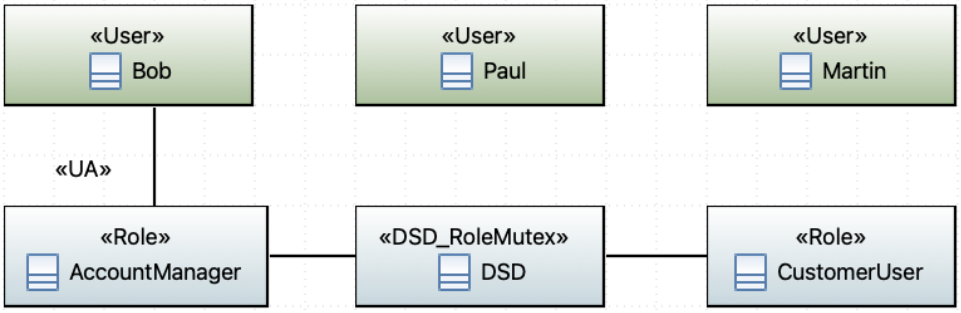

Figure 2.11: Users and Role assignment

The B specification issued from the SecureUML model is dedicated to grant or forbid functional operations given the set of roles that are activated by a user. For example, if *Paul* is a *CustomerUser*, therefore he can only read his personal data by calling basic getters of class *Customer*. The other operations (modification, creation, etc) are forbidden to him. In order to translate the security model, our approach follows two steps: (*i*) propose a "stable" formalization of a RBAC meta-model, then (*ii*) translate a given security model and inject it into the formalization of the meta-model. Each operation of the functional model is encapsulated in a secure operation checking that the current user is allowed (or not) to call this operation.

2.2.3.1 Formal modeling of user assignments and role activation

Figure 2.12 presents the structural part of the B model (Machine *UserAssignments*) that defines the right side for Figure 2.1. It refers to: *ROLES, USERS* and *SESSIONS*. The assignment of roles to users is defined with relation *roleOf*. Contrary to users and roles, which are explicitly represented with sets, sessions are defined by means of a relation between users and roles. We consider that a user cannot open several sessions in the system. When a user u belongs to the domain of relation *Session*, thus a session is created for him and $Session[\{u\}]$ gives the

set of roles that are activated by user u. Other RBAC concepts, such as role hierarchy, static separation of duties and dynamic separation of duties, are also defined by this specification. Variable *currentUser* is useful during the animation of the model because it allows us to identify the operations that are being executed by a given user. Note that we do not consider concurrent accesses during animation.

MACHINE
UserAssignments
SETS
ROLES ; *USERS*
VARIABLES
roleOf, *Roles_Hierarchy*, *currentUser*, *SSD_mutex*, *DSD_mutex*, *Session*
INVARIANT
`/* Typing invariants */`
$currentUser \in USERS$
$\wedge\ roleOf \in USERS \rightarrow \mathbb{P}\ (ROLES)$
$\wedge\ Roles\_Hierarchy \in ROLES \leftrightarrow ROLES$
$\wedge\ Session \in USERS \leftrightarrow ROLES$
$\wedge\ SSD\_mutex \in \mathbb{P}_1\ (ROLES) \leftrightarrow \mathbb{N}_1$
$\wedge\ DSD\_mutex \in \mathbb{P}_1\ (ROLES) \leftrightarrow \mathbb{N}_1$

Figure 2.12: Users assignment: typing invariants

Figure 2.13 provides some security invariants of machine *UserAssignments*. For example, cycles in a role hierarchy are not allowed even if this is graphically possible. In our approach, B4MSecure creates the valuations of the various data structures and then ProB is applied to verify that security invariants are preserved. The second invariant guarantees the conformance between role assignments and role activations. It means that, in a session, a user can only activate roles among those that are assigned to him. The other invariants guarantee SSD and DSD constraints. They are somehow redundant: the former refers to relation *roleOf* and the latter refers to relation *Session*.

Machine *UserAssignments* provides several utility operations that are useful during the animation. Figure 2.14 gives the B specifications of operations *safeConnect*, *addSafeRole* and *setCurrentUser*. Operation *safeConnect* creates a session to a given user in which a set of roles is activated. This operation is done under several conditions: (*i*) the user is not concerned by any existing session, (*ii*) if a role r_1 is a super-role of a role r_2, therefore the user can activate r_1 or r_2 but not both of them $\{r_1, r_2\}$, and (*iii*) the DSD constraints are preserved. Operation *addSafeRole* adds a new role to a given user and preserves SSD constraints. This operation is useful when roles are created dynamically because of the evolution of the functional state. For example when a customer is created, role *CustomerUser* must be added to the corresponding user. Operation *setCurrentUser* selects the user who will execute the system.

/* No cycles in role hierarchy */
$\wedge\ (Roles\_Hierarchy)^+ \cap \mathbf{id}(ROLES) = \emptyset$

/* Conformance of role assignments and role activation */
$\wedge\ \forall\ (uu).(uu \in USERS \wedge uu \in \mathbf{dom}(Session) \Rightarrow Session[\{uu\}] \subseteq roleOf(uu))$

/* Static separation of duties */
$\wedge\ \forall\ nn.(nn \in \mathbb{N}_1 \wedge nn \in \mathbf{ran}(SSD\_mutex) \Rightarrow nn \geq 2)$
$\wedge\ \forall\ rs.(rs \in \mathbb{P}_1\ (ROLES) \wedge rs \in \mathbf{dom}(SSD\_mutex) \Rightarrow \mathbf{card}(rs) \geq SSD\_mutex(rs))$
$\wedge\ \forall\ uu.(uu \in USERS \wedge uu \in \mathbf{dom}(roleOf) \Rightarrow$
$\quad \forall\ rs.(rs \in \mathbb{P}_1\ (ROLES) \wedge rs \in \mathbf{dom}(SSD\_mutex) \Rightarrow$
$\quad\quad \mathbf{card}((Roles\_Hierarchy)^+[roleOf(uu)] \cup roleOf(uu) \cap rs) < SSD\_mutex(rs)$
$\quad)$
$)$

/* Dynamic separation of duties */
$\wedge\ \forall\ nn.(nn \in \mathbb{N}_1 \wedge nn \in \mathbf{ran}(DSD\_mutex) \Rightarrow nn \geq 2)$
$\wedge\ \forall\ rs.(rs \in \mathbb{P}_1\ (ROLES) \wedge rs \in \mathbf{dom}(DSD\_mutex) \Rightarrow \mathbf{card}(rs) \geq DSD\_mutex(rs))$
$\wedge\ \forall\ uu.(uu \in USERS \wedge uu \in \mathbf{dom}(Session) \Rightarrow$
$\quad \forall\ rs.(rs \in \mathbb{P}_1\ (ROLES) \wedge rs \in \mathbf{dom}(DSD\_mutex) \Rightarrow$
$\quad\quad \mathbf{card}((Roles\_Hierarchy)^+[Session[\{uu\}]] \cup Session[\{uu\}] \cap rs) < DSD\_mutex(rs)$
$\quad)$
$)$

Figure 2.13: Users assignment: security invariants

2.2.3.2 Formal modeling of permission assignments

To formally define permissions, we first apply a deep embedding approach to the entities of the model that are associated to permissions. Machine *RBAC_Model* of Figure 2.15 gives un excerpt of the B data structures that are generated by B4MSecure. Sets *ENTITIES, Attributes, Operations* and *KindsOfAtt* represent concepts of the meta-model dealing with the functional concerns. For example, set *ENTITIES* represents classes of the functional model. Relations between these sets are: *AttributeKind* (private or public), *AttributeOf* (for class attributes), *OperationOf* (for class operations), *constructorOf* (for class constructors), *destructorOf* (for class deletion), *setterOf* (for attribute setters) and *getterOf* (for attribute getters). The other data structures of machine *RBAC_Model* define RBAC concerns. The machine also includes the functional model and machine *UserAssignments*, and promotes operations *safeConnect*, *disconnect* and *setCurrentUser*.

Relation *isPermitted* defines pairs $(r \mapsto o)$ that are calculated from the RBAC model in order to establish for every role r and operation o whether r is allowed to call o or not. For example, *CustomerUserPerm1* leads to two pairs $(CustomerUser \mapsto Customer\_GetName)$ and $(CustomerUser \mapsto Customer\_GetAddress)$. Indeed, this permission is related to role *Cus-*

safeConnect(*user*, *roleSet*) =
  PRE
    $user \in USERS \wedge user \notin \mathbf{dom}(Session)$
    $\wedge\ roleSet \in \mathbb{P}_1(ROLES) \wedge roleSet \subseteq roleOf(user)$
    `/* avoid hierarchical redundancy in the roleSet */`
    $\wedge\ \forall (r1,r2).(r1 \in roleSet \wedge r2 \in roleSet \wedge r1 \neq r2 \Rightarrow r2 \notin \mathbf{closure1}(Roles\_Hierarchy)[\{r1\}])$
    `/* avoid DSD violation */`
    $\wedge\ \forall rs.(rs \in \mathbb{P}_1(ROLES) \wedge rs \in \mathbf{dom}(DSD\_mutex) \Rightarrow \mathbf{card}((Roles\_Hierarchy)^+[roleSet] \cup roleSet \cap rs) < DSD\_mutex(rs))$
  THEN
    $Session := Session \cup (\{user\} \times roleSet)$
  END;

addRoleSafe(*user*, *role*) =
LET *newRoles* **BE** $newRoles = roleOf(user) \cup \{role\}$ **IN**
  PRE
    $user \in USERS$
    $\wedge\ role \in ROLES \wedge role \notin (roleOf(user) \cup (Roles\_Hierarchy^{-1})^+[roleOf(user)])$
    `/* avoid SSD violation */`
    $\wedge\ \forall rs.(rs \in \mathbb{P}_1(ROLES) \wedge rs \in \mathbf{dom}(SSD\_mutex) \Rightarrow \mathbf{card}(((Roles\_Hierarchy)^+[newRoles] \cup newRoles) \cap rs) < SSD\_mutex(rs))$
  THEN
    $roleOf := (\{user\} \lhd roleOf) \cup \{(user \mapsto newRoles)\}$
  END
END;

setCurrentUser(*user*) =
  PRE
    $user \in USERS \wedge user \neq currentUser \wedge user \in \mathbf{dom}(Session)$
  THEN
    $currentUser := user$
  END ;

Figure 2.14: Users assignment: utility operations

tomerUser and refers to an *EntityAction* of type *read*, and to class *Customer*. Two attributes are defined in this class (*Name* and *Address*) and their getters are respectively *Customer_GetName* and *Customer_GetAddress*. The calculation of relation isPermitted is carried out in the initiali-

sation clause of machine *RBAC_Model* based on the other B data of this machine.

MACHINE *RBAC_Model*
INCLUDES *Functional_Model*, *UserAssignements*
PROMOTES *safeConnect*, *disconnect*, *setCurrentUser*
SETS
ENTITIES ; Attributes ; Operations ; KindsOfAtt = {*public*, *private*};
PERMISSIONS ; ActionsType = {*read*, *create*, *modify*, *delete*, *fullAccess*}
CONSTANTS
AttributeKind, *AttributeOf*, *OperationOf*,
constructorOf, *destructorOf*, *setterOf*, *getterOf*,
PermissionAssignement, *EntityActions*, *MethodActions*,
VARIABLES
isPermitted
PROPERTIES
$AttributeKind \in Attributes \rightarrow KindsOfAtt \wedge$
$AttributeOf \in Attributes \rightarrow ENTITIES \wedge$
$OperationOf \in Operations \rightarrow ENTITIES \wedge$
$constructorOf \in Operations \rightarrowtail ENTITIES \wedge$
$destructorOf \in Operations \rightarrowtail ENTITIES \wedge$
$setterOf \in Operations \rightarrowtail Attributes \wedge$
$getterOf \in Operations \rightarrowtail Attributes \wedge$
$setterOf \cap getterOf = \emptyset \wedge$
$PermissionAssignement \in PERMISSIONS \rightarrow (ROLES \times ENTITIES) \wedge$
$EntityActions \in PERMISSIONS \nrightarrow \mathbb{P}(ActionsType) \wedge$
$MethodActions \in PERMISSIONS \nrightarrow \mathbb{P}(Operations) \wedge$
INVARIANT
$isPermitted \in ROLES \leftrightarrow Operations$

Figure 2.15: Structural part of RBAC machine

2.2.3.3 Formal modeling of secure operations

B4MSecure produces for every functional operation, a secured operation that verifies (using a security guard) whether the current user is allowed to call the functional operation. The secure operation also verifies the authorization constraints, if they are defined in the underlying permissions, and updates the assignment of roles when required.

Figure 2.16 shows the secure operations associated to *Account_transferFunds* and *Customer_NEW*. The security guard is defined in clause *SELECT*. It verifies that the functional operation belongs to set *isPermitted*[*currentRoles*], where definition *currentRoles* refers to the roles activated by *currentUser* (in a session) as well as their super-roles:

$$
\begin{array}{|l}
\hline
\mathit{currentRoles} == \\
\qquad \mathit{Session}[\{\mathit{currentUser}\}] \cup \\
\qquad \mathbf{ran}(\mathit{Session}[\{\mathit{currentUser}\}] \lhd (\mathit{Roles\_Hierarchy})^{+}) \\
\hline
\end{array}
$$

secure_Account_transferFunds(*aAccount*, *NB*, *m*) =
PRE
aAccount ∈ *Account* ∧ *NB* ∈ ℕ ∧ *m* ∈ ℕ$_1$ [∧ . . .]
THEN
SELECT
*Account_transferFunds_* ∈ *isPermitted*[*currentRoles*]
∧ (*CustomerUser* ∈ *currentRoles* ⇒ *AccountOwner*(*aAccount*) = *currentUser*)
THEN
Account_transferFunds(*aAccount*, *NB*, *m*)
END
END;

secure_Customer_NEW(*aCustomer*, *theAccount*) =
PRE
aCustomer ∈ *CUSTOMER* ∧ *theAccount* ∈ $\mathcal{F}$(*Account*) [∧ . . .]
THEN
SELECT
*Customer_NEW_* ∈ *isPermitted*[*currentRoles*]
THEN
Customer_NEW(*aCustomer*, *theAccount*) ||
addRoleSafe(*aCustomer*, *CustomerUser*)
END
END;

Figure 2.16: Operational part of RBAC machine

The security guard of *secure_Account_transferFunds* is strengthened with the authorization constraint of permission *CustomerUserPerm2*. For every permission p associated to a role r and a constraint c, the tool adds guard $(r \in \mathit{currentRoles} \Rightarrow c)$ to all operations that are concerned with p. Note that in the graphical model the constraint is directly written in B. In this case the constraint is: *AccountOwner*(*aAccount*) = *currentUser*. Regarding operation *secure_Customer_NEW*, it is not concerned with an authorization constraint. But, as class *Customer* and role *CustomerUser* are aligned, the operation applies *addSafeRole* to update relation *roleOf* when a customer is created.

These two operations show how B4MSecure takes into account the relationship between functional and security models. In one case the permission is granted to users when some functional conditions hold and in the other case the creation of some objects may grant new permissions to users since they may acquire additional roles.

2.3 Animation and dynamic analyses

The example in the previous section highlights the need for dynamic analyses of both functional and security models. When the security policy refers to functional elements, a dynamic analysis should not only cover the RBAC functions, but also take into account the behaviour of the functional model. Dynamic analyses can take two forms: test and model-checking. Tests correspond to the execution of a sequence of actions on the various models, or on their implementations. The test sequence is either defined by the security policy designer, possibly on the basis of use cases, or it may be the output of a test generation tool.

However, tests can only check a limited number of behaviours. When absolute guarantees are required, such as ensuring that all threats are handled, verification techniques, such as model-checking and theorem proving, should be considered. Theorem proving can show that constraints are satisfiable, or establish that some property, like SoD, is an invariant of the model. However, as we are interested by the reachability of particular states that may open breaches and favour security threats, theorem proving is not an efficient technique. We apply theorem proving to guarantee that the functional model is correct regarding its invariants as discussed in Section 2.2.2, and also to guarantee that the pre-established security operations are correct with respect to the RBAC constraints (*i.e.* Role hierarchy, SSD, DSD, etc).

2.3.1 Animation in B4MSecure

In order to favour dynamic analyses in one integrated framework, B4MSecure applies ProB Java API as presented in Figure 2.17.

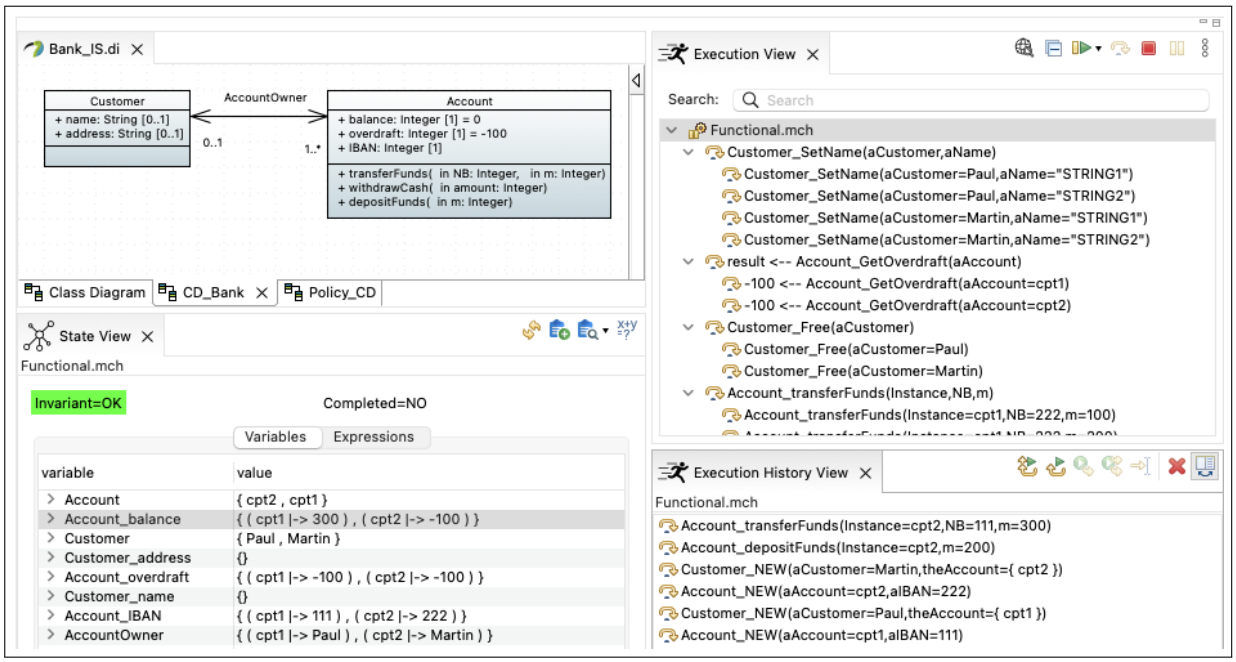

Figure 2.17: Animation in B4MSecure

Note that the B specifications that are extracted by the tool can also be verified outside B4MSecure using ProB standalone. The State View (bottom-left) gives the values of the B variables in the current state, *i.e.* after animating the sequence of the History View (bottom-right). The Execution View (top-right) shows the B operations that can be triggered in the current state. The content of these views is computed by the Java API of ProB; B4MSecure just provides some convenient actions to ensure animation and/or model-checking via the API.

Figure 2.17 shows a sequence of functional operations that creates customers Paul and Martin, as well as their respective accounts cpt_1 and cpt_2. An amount of 200€ is added to Martin's account and then 300€ are transferred from this account to Paul's account.

Account_NEW(cpt$_1$, 111) ;
Customer_NEW(Paul,{cpt$_1$}) ;
Account_NEW(cpt$_2$, 222) ;
Customer_NEW(Martin,{cpt$_2$}) ;
Account__depositFunds(cpt2, 200) ;
Account__transferFunds(cpt2, 111, 300)

$\downarrow\downarrow\downarrow$

Account_balance = $\{(cpt_1 \mapsto 300), (cpt_2 \mapsto -100)\}$
AccountOwner = $\{(cpt_1 \mapsto Paul), (cpt_2 \mapsto Martin)\}$
Account_IBAN = $\{(cpt_1 \mapsto 111), (cpt_2 \mapsto 222)\}$
Account_overdraft = $\{(cpt_1 \mapsto -100), (cpt_2 \mapsto -100)\}$

Figure 2.18: Valid functional sequence and its target state

The scenario of Figure 2.18 that is animated in B4MSecure (Figure 2.17) corresponds to a normal use case of the IS. Playing with functional scenarios shows that use cases are feasible with the current specification, and helps identify missing steps in the use cases or the specification. A similar animation can be performed by calling the secured version of the use case (machine *RBAC_Model*). This eases the understanding and validation of the security policy and shows that the security policy does not prevent the execution of functional use cases.

2.3.2 Testing the security policy

Having an operational formal model of the security policy, it becomes possible to carry out security tests, by testing permissions and/or prohibitions. For example, when a test consists of reading an object, one can simply check that the getter (or a reading operation) is allowed (or not) in a given state. Several testing objectives can be addressed when testing a secure operation: with a role that does not have access to this operation, without satisfying an authorization constraint, or outside the operation precondition.

In order to perform security tests, we structure the animation of secure operations, according to their permissions. We consider three parts: preamble, nominal/robustness and call. The "preamble" sequence leads to a state allowing the execution of a permitted operation. The

"nominal/robustness" varies the users and the roles that are concerned by the permission granting access to the operation. The "nominal" sequence corresponds to a positive test, and sequence "robustness" presents a slight variation compared to the nominal case, which must lead to a failure (negative test). Finally, the "call" part applies the operation that we want to test in order to see that it can be effectively called if it is authorized. For example, if we want to test operation *depositFunds*, granted by permission *CustomerUserPerm2* to role *CustomerUser*, we consider the sequences of Figure 2.19.

```
preamble =
BEGIN
   safeConnect(Bob, {AccountManager}) ;
   setCurrentUser(Bob) ;
   secure_Account_NEW(cpt1, 111) ;
   secure_Customer_NEW(Paul,{cpt1}) ;
   secure_Account_NEW(cpt2, 222) ;
   secure_Customer_NEW(Martin,{cpt2})
END ;
```

```
nominal =
BEGIN
   safeConnect(Paul, {CustomerUser}) ;
   setCurrentUser(Paul) ;
END ;
robustness =
BEGIN
   safeConnect(Martin, {CustomerUser}) ;
   setCurrentUser(Martin)
END ;
```

```
call =
BEGIN
   secure_Account__depositFunds(cpt1,500)
END
```

Figure 2.19: Structuring security tests

Positive tests verify that a user can execute the permitted operation if he/she activates the corresponding role and satisfies the functional preconditions as well as the authorization constraint. For example, sequence « preamble ; nominal ; call » is a positive test and the resulting state is represented with the object diagram of Figure 2.20. In this sequence, Paul deposits money into his own account.

Figure 2.20: Initial object diagram

Positive tests that fail during the animation may reveal some defects in the definition of security rules. For example, a use case where the account manager modifies the allowed overdraft of an account should be feasible. However, the following test sequence, which is therefore a positive test, cannot be animated:

safeConnect(Bob, {AccountManager}) ;
setCurrentUser(Bob) ;
secure_Account_NEW(cpt_1, 111) ;
secure_Account_setOverDraft(cpt_1, -500)

Operation *Account_setOverDraft* is a basic operation, generated by B4MSecure in order to modify the value of attribute *OverDraft* of class *Account*; but there is no permission granting access to this operation. Failure of this test reveals that a permission has been forgotten.

Regarding negative tests, they are close to positive tests, but they invalidate one of the elements of the permission: role, precondition or authorization constraint. Contrary to positive tests, a negative test cannot fully run, since it contains an illegal call to a secure operation. A negative scenario, should not be allowed by the animator. For example, sequence « preamble ; robustness ; call » is a negative test that invalidates the authorization constraint. In this sequence, Martin tries to deposit money into an account that does not belong to him.

2.3.3 Attack scenarios

The advantage of theorem proving and testing, discussed in the previous sections, is that when looking for threats, the security analyst has the guarantee that flaws are not issued from invariant violations, but rather from the functional or the security logic. In this sense, the identification of attack scenarios is mainly a validation task, which can be done by model-checking. Indeed, an exhaustive model exploration may exhibit a malicious sequence of operations leading to a given state (where a property holds) from which a bad action may be done.

To show a possible malicious scenario based on our simple example, we start the exploration from a normal state, that of Figure 2.20, reached by sequence « preamble ; nominal ; call ». In this state *Paul* is a customer and owns account cpt_1 whose balance is equal to 500. *Bob* as *AccountManager* cannot execute operations *transferFunds* or *withdrawCash* on cpt_1. The answer to a static query such as "Is Bob able to transfer funds from Paul's account?" would be NO, since the permission given to a manager on class *Account* only allows instance creation. However, the good question should be "Is there a sequence of operations that can be executed by Bob in order to become able to transfer funds from Paul's account?". To answer the question we use the model-checking feature of ProB to explore the state space and find states that satisfy property: $\mathit{AccountOwner}(\text{cpt}_1) = \text{Bob}$. We are therefore looking for a sequence of operations executed by Bob allowing him to become the owner of cpt_1. After finding the state we use ProB to verify whether it is possible to come back into a state where cpt_1 is owned by *Paul*. The objective is to check if the attacker can execute the reverse actions in order to hide his attack.

Sequence of Figure 2.21 has been exhibited by ProB after exploring more than 36000 reachable states. We structure it in four steps. In **`step 1`** *Bob* adds himself to the system as a customer. As the creation of a customer requires at least one account, *Bob* creates the fictive account cpt_3 and then he calls operation *secure_Customer_NEW*. In **`step 2`**, the attacker becomes the owner of cpt_1. To this purpose he must first remove the link between *Paul* and cpt_1.

But, in the functional model a customer must have at least one account, consequently operation *secure_Customer_RemoveAccount*(*Paul*, $\{cpt_1\}$) is possible only if *Paul* has another account. For this reason, *Bob* creates another fictive account cpt_4 and adds it to *Paul*'s accounts. The last action of **step 2** reaches the malicious state where *Bob* is the owner of cpt_1. Finally, **step 3** realizes the attack and **step 4** brings the system back to a normal state.

```
/* step 1:  create customer Bob */
  safeConnect(Bob, {AccountManager}) ;
  setCurrentUser(Bob) ;
  secure_Account_NEW(cpt3, 333) ;
  secure_Customer_NEW(Bob,{cpt3}) ;
/* step 2:  get the ownership of Paul's Account */
  secure_Account_NEW(cpt4, 444) ;
  secure_Customer_AddAccount(Paul,{cpt4}) ;
  secure_Customer_RemoveAccount(Paul,{cpt1}) ;
  secure_Customer_AddAccount(Bob,{cpt1}) ;
/* step 3:  attack */
  disConnect(Bob) ;
  safeConnect(Bob, {CustomerUser}) ;
  secure_Account_transferFunds(cpt1, 333, 500) ;
/* step 4:  hide the attack */
  disConnect(Bob) ;
  safeConnect(Bob, {AccountManager}) ;
  secure_Customer_RemoveAccount(Bob,{cpt1}) ;
  secure_Customer_AddAccount(Paul,{cpt1}) ;
  secure_Customer_RemoveAccount(Paul,{cpt4}) ;
  secure_Customer_Free(Bob)
```

Figure 2.21: Malicious scenario

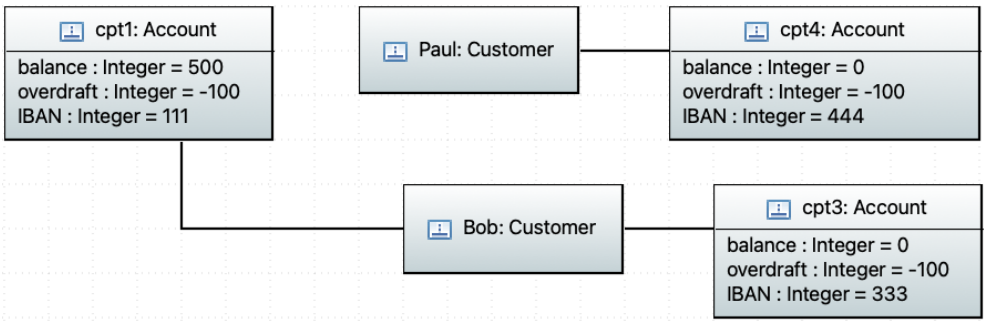

Figure 2.22: Malicious state reached after the execution of steps 1 and 2

This malicious scenario can be countered by enhancing the functional model and/or the security model. If the analyst assumes that the flaw is favored by the functional logic, one possible solution would be to introduce the following invariant:

$$\boxed{Account\_\_balanceValue \neq 0 \Rightarrow AccountOwner[\{Instance\}] \neq \emptyset}$$

In fact, operation $secure\_Customer\_RemoveAccount(Paul, \{cpt_1\})$ is the dangerous operation. This invariant means that accounts whose balance is not equal to zero must be owned by a customer. By introducing this invariant, several functional operations must be corrected and proved, such as: *Customer_RemoveAccount* and *Account_SetBalance*. In other words, to remove the ownership relation between cpt_1 and *Paul*, the account of *Paul* must be empty.

If the analyst assumes that the flow is favored by the security logic, a possible solution would be to limit the scope of permission *AccountManagerPerm1* because it currently grants a full access to role *AccountManager* on customer's data, including the deletion of his accounts.

2.4 Discussion and conclusion

One of the advantages of B4MSecure (Akram Idani and Ledru, 2015) is its extensibility; besides an extension to Event-B has been proposed in (Wakrime et al., 2018) and an extension covering other security features such as organisations and contextual rules has been defined by Yangui (2016). Not only the translation of functional models is built on existing UML-to-B works that are approved by the formal methods community and that are combined together in one unifying framework, but also the translation of RBAC models can be configured leading to various translations. This chapter has been focused on one possible translation providing an overview of our Formal Model-Driven Security approach.

The major motivation behind the tool is that MDS (Basin et al., 2006) advocates for the separation of concerns principle and suggests the validation of functional and security models in isolation. However, access control rules often include authorisation constraints, which requires dynamic analyses dealing with elements of both models. Our proposal is built on the B method and allows the validation and verification of RBAC policies involving constraints. This work has been applied and experimented in several research projects, the major ones are: ANR Selkis (2008-2012) and NExTRegio of IRT Railenium (2015-2019). The first project addresses medical information systems and the second project deals with railway applications.

ANR Selkis (2008-2012). The first versions of B4MSecure have been developed during this project, and have involved two PhD students (N. Qamar and M.-A. Labiadh) and two M2 students that I co-supervised with my colleague Y. Ledru. In (Ledru et al., 2015a) we reported on the application of B4MSecure to Res@mu, a case study issued from this project. Res@mu is a medical IS developed by Ifremmont[7]. It supports every stage of a medical urgency, from the

[7] Ifremmont (http://www.ifremmont.com/): Institut Français de recherche en Médecine de Montagne.

phone call to an emergency call center, to the management of emergency teams during a mission. The starting point of this study was a set of UML class diagrams and use cases describing the data structures and main functionalities of the IS. These include 77 classes and more than 100 use cases. The application of B4MSecure was focused on 12 classes. We discarded classes that do not influence medical acts, our security target, like those describing rescue vehicles or their base station, or the motivation and circumstances of emergency missions. A total of 34 positive tests and 87 negative test have been defined and played against the models using ProB. They helped us incrementally define the security policy, following a TDD approach. Thanks to these tests, errors have been identified in functional use cases, in the initial version of our access control rules or in manually defined assertions of B models. We believe that the success of these tests gives reasonable guarantees in the quality of our security policy. We have also defined 4 attack scenarios, and several associated variants. They correspond to 7 positive tests and 13 negative ones. They all try to compromise the confidentiality of medical acts. In these scenarios, users with high privileges (*e.g.* team doctor or regulator) try to get access to the medical acts of a patient by joining the intervention team during or after the intervention. All attacks failed, which increased our confidence in the security policy. These attacks showed the usefulness of protecting not only the security target, but also related classes involved in authorisation constraints. However, the scenarios describing these attacks are generally complex and ProB was not able to compute the enabledness for most of them (18/20 tests), experiencing time-outs or memory errors. We had to break these tests down into smaller test steps that were played in sequence. The next chapter discusses some solutions to these issues, such as combining abstraction and theorem proving, to automate the extraction of attack scenarios.

NExTRegio (2015-2019). Papers (Ledru et al., 2019) and (Wakrime et al., 2018) show how B4MSecure has been used in the railway field. During this collaboration with IFSTTAR[8] and IRT Railenium[9] we proposed a modelling approach for railway signalling rules being inspired by MDS and IS security. The approach models the agents that perform railway operations and the conditions that must be satisfied before performing these operations. These models are expressed in SecureUML diagrams enhanced with B assertions, and then translated, using B4MSecure, into B machines. ProB is applied to verify the model via model-checking and animation in order to assess the reachability of desired states, and verify the absence of accidents. Furthermore, the approach proceeds by introducing human errors, checking their consequences, and deploying counter-measures. For this case study the separation of concerns principle was useful to define first an uncontrolled model only governed by the laws of physics, where accidents may happen, and second a model controlled by signalling rules, where bad things should not happen. This is similar to the distinction made in secure IS and MDS between data and associated functions, described in a so-called "functional" model, and the permissions that rule the accesses of users to these data, described in a "security" model. Each operation of the un-

[8] IFSTTAR: Institut français des sciences et technologies des transports, de l'aménagement et des réseaux, https://www.ifsttar.fr/

[9] IRT Railenium: https://railenium.eu

controlled model has a controlled version in the control model. This version adds guards to check the relevant permissions. For example, trains, tracks, lights and their associated operations correspond to the uncontrolled model. Train drivers and traffic agents must follow the rules that constrain the call to these operations, which is a kind of role-based access control. Having this formal specification, the existence of accidents is assimilated to insider threats. Indeed, the verification of signalling rules assumes that users follow the rules, *i.e.* they only access operations that are permitted, and follow authorisation constraints. These assumptions are not valid in the case of human errors, for example (1) the train driver can overlook an off light and enter a forbidden track portion or (2) the traffic agent can switch on a light ignoring the corresponding safety constraints. Such human errors can be the consequence of tiredness. In 2016, Infrabel, the Belgian railway company, reported that 91 trains (out of 1,3 million) ignored a red light (Infrabel, 2017).

Chapter 3

Looking for malicious behaviours

Contents

This chapter is an update of these two papers:

Amira Radhouani, Akram Idani, Yves Ledru, and Narjes Ben Rajeb. Symbolic Search of Insider Attack Scenarios from a Formal Information System Modeling. *LNCS Transactions on Petri Nets and Other Models of Concurrency*, 10:131–152, 2015. URL https://doi.org/10.1007/978-3-662-48650-4_7.

Amira Radhouani, Akram Idani, Yves Ledru, and Narjes Ben Rajeb. Extraction of insider attack scenarios from a formal Information System Modeling. In *5th International Workshop on Formal Methods for Security (FMS)*, 2014.

The early detection of potential threats during the modelling and design phase of a Secure Information System is required because it favours the design of a robust access control policy and the prevention of malicious behaviours during system execution. This paper deals with internal attacks which can be made by people inside the organization.

Such attacks are difficult to detect because insiders have authorized system access and also may be familiar with system policies and procedures. We are interested in finding attacks which conform to the access control policy, but lead to unwanted states. These attacks are favoured by policies involving authorization constraints, which grant or deny access depending on the evolution of the functional Information System state. In this context, we propose to model functional requirements and their Role Based Access Control (RBAC) policies using B machines and then to formally reason on both models. In order to extract insider attack scenarios from these B specifications, our approach first investigates symbolic behaviours. Then, the use of a model-checking tool allows to exhibit, from a symbolic behaviour, an observable concrete sequence of operations that can be followed by an attacker. In this chapter, we show how this combination of symbolic analysis and model-checking allows one to find out such insider attack scenarios.

3.1 Running Example

In this section we use a running example issued from (Basin et al., 2009) and which deals with a SecureUML model associated to a functional UML class diagram.

3.1.1 Functional model

The functional UML class diagram (presented in Figure 3.1) describes a meeting scheduler dedicated to manage data about two entities: Persons and Meetings.

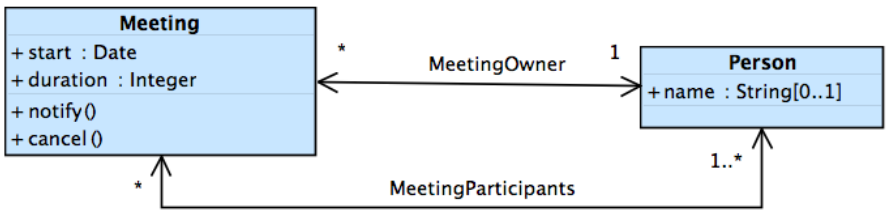

Figure 3.1: Functional model of meeting scheduler system

A meeting has one and only one owner (association *MeetingOwner*), a list of participants (association *MeetingParticipants*), a duration, and a starting date. A person can be the owner of several meetings and may participate to several meetings. Operations *notify* and *cancel* are user-defined, and allow respectively to send messages to participants and to delete a meeting after notifying their participants by e-mail. Constructors, setters and getters are implicitly defined for both classes and both associations.

3.1.2 Access control rules

The access control model is given in Figure 3.2 using the secureUML syntax. It features three different roles:

- **SystemUser**: defines persons who are registered on the system and then have permission **UserMeetingPerm** which allows them to create and read meetings. Deletion and modification of meetings (including operation *cancel*) are granted to system users by means of permission **OwnerMeetingPerm**, featuring an authorization constraint checking that the user who tries to run these actions is the meeting owner.
- **Supervisor**: defines system users with more privileges because they can run actions notify and cancel on any meeting even if they are not owners.
- **SystemAdministrator**: having a full access on entity Person, an administrator manages system users. Full access grants him the right to create a new person, remove or modify an existing one. Furthermore, a system administrator has only a read access on meetings: he is not expected to create or modify meetings.

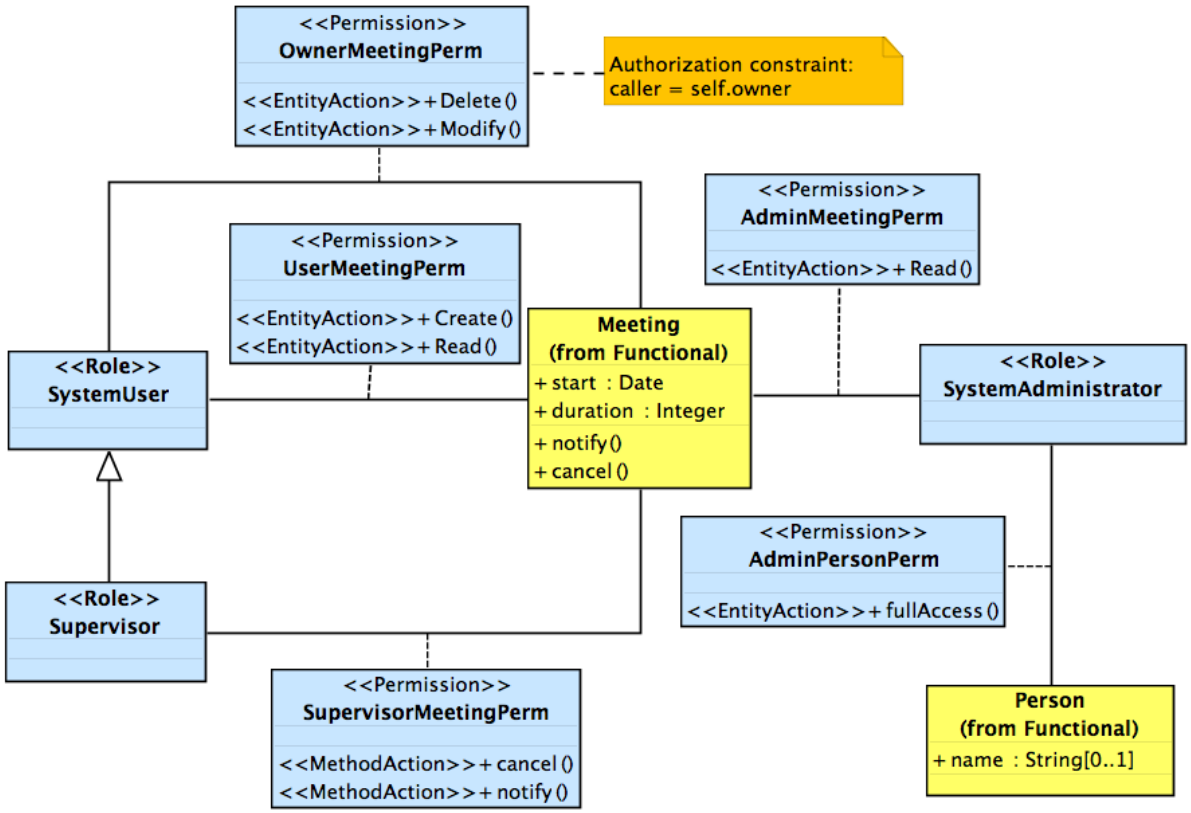

Figure 3.2: Security model of meeting scheduler system

3.1.3 Validation

This example is intended to be validated in (Basin et al., 2009) based on a set of static queries that investigate a given system state in order to grasp some useful information like "*which user*

can perform an action on a concrete resource in a given state".

Authorization constraint associated to **OwnerMeetingPerm** requires information from the functional model because it deals with the *MeetingOwner* association. In the remainder, we consider three users John, Alice and Bob such that user assignments are as defined by Figure 3.3. We also consider a given initial state in which Alice is owner of meeting m_1, Bob is a participant of m_1. In such a state, the above static query establishes that only Alice is allowed to modify or delete m_1 because she is the owner of m_1.

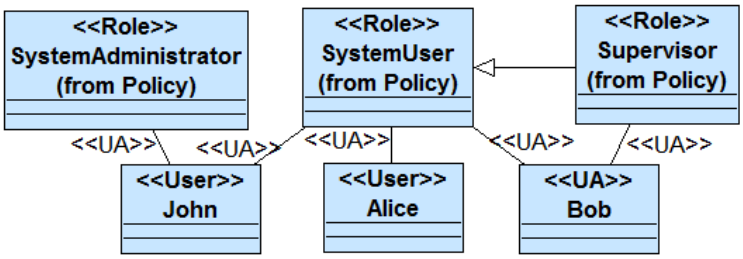

Figure 3.3: Users assignement

In (Ledru et al., 2014) a dynamic analysis approach based on animation of a formal specification showed that validation should not only be based on a given static state, but should search for sequences of actions modifying this state and breaking the authorization constraint. For example, starting from the above state, a static query would only report that John, and also Bob, can't modify m_1 because none of them satisfies the authorization constraint. A dynamic analysis would ask if there exists a sequence of operations enabled by John, or Bob, that allows them to modify m_1. This paper contributes towards automatically finding these malicious sequences. To perform these analysis, we applied the B4MSecure tool to the UML and SecureUML diagrams and generated a B specification counting 946 lines. This tool generates automatically a specification for all basic functional operations, which is enriched manually by some user-defined operations (*i.e.* cancel, notify).

3.2 Dynamic analysis

3.2.1 Trace Semantics for B Specifications

In order to find malicious behaviours of an operational secure IS model, we rely on the set of finite observable traces of our B specifications. Indeed, B specifications can be approached by means of a trace semantics composed of an initialization substitution $init$, a set of operations $\mathcal{O}$ and a set of state variables $\mathcal{V}$. We note val a possible state predicate allowed by the invariant and op an operation from $\mathcal{O}$. A functional behaviour is an observable sequence $\mathcal{Q}$

$$\mathcal{Q} \mathrel{\hat{=}} init \ ; \ op_1 \ ; \ op_2 \ ; \ \ldots \ ; \ op_m$$

such that $\forall i.(i \in 1..m \Rightarrow op_i \in \mathcal{O})$ and there exists a sequence $\mathcal{S}$ of state predicates which does not violate invariant properties:

$$\mathcal{S} \mathrel{\hat{=}} val_0 \ ; \ val_1 \ ; \ \ldots \ ; \ val_m$$

in which val_0 is an initial state, and op_i is enabled from state val_{i-1} and state val_i is reached by op_i, starting from state val_{i-1}.

The security model filters functional behaviours by analysing access control premises which are triplets (u, R, c) where u is a user, R is a set of possible roles assigned to u, and c is an authorization constraint. An observable secure behaviour is a sequence $\mathcal{Q}$, where for every step i, premise (u_i, R_i, c_i) is valid (expressed as $(u_i, R_i, c_i) \models true$). This means that roles R_i activated by user u_i grant him/her the right of running operation op_i and if a constraint c_i exists, then it must be satisfied. The following premises sequence $\mathcal{P}$ must be valid for $\mathcal{Q}$:

$$\mathcal{P} \mathrel{\hat{=}} (u_1, R_1, c_1) \ ; \ (u_2, R_2, c_2) \ ; \ \ldots \ ; \ (u_m, R_m, c_m)$$

3.2.2 Tools to exhibit behaviours from B specifications

Model-checking and symbolic proof techniques are of interest in order to exhibit a relevant behaviour from an operational B specification. Proof techniques deal with infinite systems and can prove constraint satisfiability, or establish that some operation can be enabled from an abstract state predicate. Model-checking is based on model exploration of finite systems, and can be used to find a sequence of actions leading to a given state or property. In our approach, we combine both techniques in order to overcome their shortcomings: complexity of proofs for the first one, and state explosion for the second one. In this sub-section, we illustrate both tools.

Model checking and animation (the ProB tool). ProB (Leuschel and Butler, 2003) is an animation and a model-checker of B specifications that explores the concrete state space of the specification and generates accessibility graphs. Then, every predicate val_i (where $i \in 0 \ldots m$) of sequence $\mathcal{S}$ is a valuation of variables issued from $\mathcal{V}$. For example, considering $\mathcal{V} = \{person, meeting, meetingOwner, meetingParticipants\}$ and starting from an initial state val_0 such that:

$$\begin{aligned} val_0 \mathrel{\hat{=}} & \ person = \emptyset \\ & \wedge meeting = \emptyset \\ & \wedge meetingOwner = \emptyset \\ & \wedge meetingParticipant = \emptyset \end{aligned}$$

and having $\mathcal{O} = \{personNew, meetingNew, meetingAddParticipants\}$, the scenario of table 3.1 is successfully animated using ProB. Column "reached states" gives only modified B variables from the previous step.

In step 1, the tool animates operation $personNew$ which modifies variable person (initially equal to $\emptyset$) and this action was performed by user John using role SystemAdministrator without

need of authorization constraint. In step 4, the tool adds participant Bob to the meeting m_1 by animating operation meetingAddParticipants, after verification that authorization constraint is True for Alice using role SystemUser. Indeed, Alice is the owner of m_1.

step	Sequence $\mathcal{Q}$	Reached states $\mathcal{S}$	RBAC premises $\mathcal{P}$
1	personNew	person={Alice}	John SystemAdministrator no constraint
2	personNew	person={Alice, Bob}	John SystemAdministrator no constraint
3	meetingNew	meeting={m_1} meetingOwner={(Alice, m_1)}	Alice SystemUser no constraint
4	meetingAddParticipants	meetingParticipants={(m_1, Bob)}	Alice SystemUser Constraint: Alice is the owner of m_1

Table 3.1: animation of a normal scenario with ProB

Symbolic proof (the GeneSyst tool). ProB is useful to animate scenarios identified during requirements analysis, or to exhaustively explore a finite subset of state space. As we are interested in finding malicious scenarios that exhibit a potential internal attack, the ProB technique may be useful only if it explores the right state space subset in the right direction, which is not obvious for infinite systems. Symbolic proof techniques, such as that of GeneSyst by Bert et al. (2005), are more interesting because they allow one to produce symbolic transition systems that represent a potentially infinite set of values and a set of predicate states. Such tools reason on the enabledness properties of an operation op from a symbolic state E and the reachability properties of a symbolic state F by op from E. In (Bert et al., 2005), three enabledness properties and three reachability properties are defined in terms of the following proof obligations, where E and F are two disjoint state predicates and x is the set of variables of the system:

(1) always enabled: $\forall x.E \Rightarrow Pre(op)$

(2) never enabled: $\forall x.E \Rightarrow \neg Pre(op)$

(3) possibly enabled ($\neg$ (1) $\wedge$ $\neg$ (2)): $\exists x.E \wedge Pre(op)$

(4) always reached: $\forall x.E \wedge Pre(op) \Rightarrow [Action(op)]F$

(5) never reachable: $\forall x.E \wedge Pre(op) \Rightarrow [Action(op)]\neg F$

(6) possibly reached ($\neg$ (4) $\wedge$ $\neg$ (5)): $\exists x.E \wedge Pre(op) \wedge \neg[Action(op)]\neg F$

Where $Pre(op)$ is a predicate representing the preconditions under which the operation op becomes feasible (Abrial and Mussat, 1998), and $Action(op)$ is a generalized substitution (Abrial, 1996) representing its action.

The first three proof obligations deal with enabledness property. Proof obligation (1) means that whenever the state E is satisfied, the precondition of the operation op is true, then op is always enabled from E. Whereas, according to the proof obligation (2), the op precondition is false when E is satisfied which means that op can never be enabled from E. Proof obligation (3) means that there exists a subset in E from which the precondition is satisfied, then op is possibly enabled from E.

In the generalized substitution theory, formula $[S]R$ means that substitution S always establishes predicate R, and $\neg[S]\neg R$ means that substitution S may establish predicate R. Hence, proof (4) means that F is always reached by the operation op from E. Proof (5) means that F is never reached by actions of op from state E. Finally, proof (6) means that state F may be reached by actions of op, when the operation precondition is true in state E. Note that reachability properties do not make sense if the enabledness property is not proved. Let us consider, for example, the functional operation $meetingNew$:

$$
\begin{array}{l}
meetingNew(m,p) \hat{=} \\
\qquad PRE \\
\qquad\quad m \notin meeting \wedge p \in person \\
\qquad THEN \\
\qquad\quad meeting := meeting \cup \{m\} \\
\qquad\quad || \; meetingOwner := meetingOwner \cup \{(m \mapsto p)\} \\
\qquad END
\end{array}
$$

This operation adds a new meeting m and links it to an owner p. If we define states E and F such that:

$$
\begin{array}{l}
E \hat{=} meetingOwner[\{m_1\}] = \emptyset \\
F \hat{=} meetingOwner[\{m_1\}] \neq \emptyset
\end{array}
$$

Therefore, proof obligation produced by GeneSyst for property (6) is successfully proved via AtelierB showing that operation $meetingNew$ when enabled from a state where m_1 does not exist and there exists at least one person in the system, may lead to a state where m_1 is created and has an owner.

As illustrated above, our work will focus on proof (6) which states the reachability of a target state from an initial one by some operations that are proved enabled from this one according to proof (3). We assume it is sufficient to decide whether an operation is potentially useful for a malicious behaviour. Proofs (4) and (5) can be used if one would like to assume that a state can never be reached, or it is always reached, by an operation.

3.2.3 Malicious Behaviour

Based on the security requirements, several operations can be identified as critical. For example, operations that perform (un)authorized modifications of meetings (*e.g.* $MeetingSetStart$) are critical because they may affect the integrity of meeting information. A malicious behaviour executed by a user u is an observable secure behaviour $\mathcal{Q}$ with m steps such that:

- op_m is a critical operation to which an authorization constraint c_m is associated.
- user u is malicious and would like to run op_m by misusing his/her roles R_u.
- val_0 : is an initial state where $(u, R_u, c_m) \models false$
- for every step i ($i \in 1..m$) premise $(u, R_u, c_i) \models true$

In other words, malicious user u is not initially allowed to execute the critical operation, but he/she is able to run a sequence of operations leading to a state from which he/she can execute this operation. In our investigation we assume that user u executes this malicious sequence without collusion with another user. This problem is left to a future work.

Section 3.1.3 gave an example where neither Bob nor John are allowed to run a modification operation, such as $meetingSetStart$ which modifies attributes of class Meeting, from the initial state due to the authorization constraint. This initial state is:

$$
\begin{aligned}
&val_0 \mathrel{\hat{=}} \\
&\quad person = \{Alice, Bob\} \\
&\quad \wedge\ meeting = \{m_1\} \\
&\quad \wedge\ meetingOwner = \{(Alice \mapsto m_1)\} \\
&\quad \wedge\ meetingParticipant = \{(m_1 \mapsto Bob)\}
\end{aligned}
$$

In the following, we denote as $init_0$ the sequence of operations leading to val_0 as presented in table 3.1. We used the model-checking facility of ProB in order to explore exhaustively the state space and automatically find a path starting from val_0 and leading to a state where operation $meetingSetStart$ becomes permitted to John. We asked ProB to find a sequence where John becomes the owner of m_1:

$$meetingOwner(m_1) = John$$

After exploring more than 1000 states, ProB found a scenario in which John executes sequentially operations personNew, personAddMeetingOwner and meetingSetStart. Indeed, John, as a system administrator, has a full access to entity Person. This permission allows him to create, modify, read and delete any instance of class Person. First, using his system administration role, he creates an instance John of class Person that corresponds to him by running operation $personNew(John)$. Then he adds meeting m_1 to the set of meetings owned by John, by running operation $personAddMeetingOwner(John, m_1)$ which is a basic modification operation of class Person. These two actions allowed him to become the owner of m_1 and then he was able to modify the meeting of Alice using his system user role.

Like all model-checking techniques, when ProB explores exhaustively the state space, it faces the combinatorial explosion problem which depends on the number of operations provided to the tool and the state space size. In order to address this problem, our approach proposes a symbolic search which finds a sequence of potentially useful operations on which the model-checker should be focused.

3.3 Symbolic Search

The proposed symbolic search is performed by an algorithm (Figure 3.4) that looks for an observable symbolic sequence $\mathcal{Q} \mathrel{\hat{=}} init_0 \ ; \ op_1 \ ; \ \ldots \ ; \ op_m$ such that operations are not instantiated. The searched sequence is executed by a user u, where (u, R_u, c_m) is not valid for a critical operation op_m in the initial state val_0 but becomes valid for state val_{m-1} where op_m can be enabled. It is a backward search algorithm, starting from the goal state val_{m-1} from which the critical operation op_m can be enabled: $val_{m-1} \hat{=} c_m \wedge Pre(op_m)$; and working backwards until the initial state val_0 is encountered. The algorithm ends when sequence $\mathcal{Q}$ is found or when all operations are verified without encountering the initial state. We consider that val_0 is a completely valuated state such as the one where Alice is the owner of m_1, and Bob is a participant to m_1. This prevents the initial state from being included in both states val_{m-1} and val_{m-2}, which would never verify the condition of the while loop.

3.3.1 Termination

Termination of our algorithm is ensured by the termination of the while loop conditions:

$$val_0 \not\Rightarrow val_{m-1} \textbf{ and } val_{m-1} \not\models false \textbf{ and } Opset \neq \emptyset$$

- $val_0 \not\Rightarrow val_{m-1}$: means that the initial state is not yet encountered. If the initial state is included in val_{m-1} ($val_0 \Rightarrow val_{m-1}$), then our algorithm concludes that a sequence is found.

- $val_{m-1} \not\models false$: means that precondition of the last computed operation is not reduced to false. If this condition becomes false, then the algorithm concludes that there is no further state enabling the last computed operation, and then it raises exception "no sequence found".

- $Opset \neq \emptyset$: means that there still exist operations that are not exploited. Every operation is called at the most once in a symbolic sequence because the algorithm iteratively removes operations from set $Opset$ ($Opset \mathrel{\hat{=}} Opset \setminus \{o_i\}$). Hence, when a sequence is found, the maximum size of a computed scenario is n such that n is the system operations number. This last condition guarantees termination when none of the previous conditions becomes false after exploiting all operations.

1. $\mathcal{Q} \hat{=} op_m$;
2. $val_{m-1} \hat{=} c_m \wedge Pre(op_m)$;
3. $val_{m-2} \hat{=} \neg val_{m-1}$;
4. $Opset \hat{=} \mathcal{O}$;
5. **while** ($val_0 \not\Rightarrow val_{m-1}$ **and** $val_{m-1} \not\models false$ **and** $Opset \neq \emptyset$) **do**
6. **choose any** $o_i \in Opset$ **where**
7. $(u, R_u, c_i) \models true \wedge$
8. $\exists x.val_{m-2} \wedge Pre(o_i) \wedge$ //PO3
9. $\exists x.val_{m-2} \wedge Pre(o_i) \wedge \neg[Action(o_i)]\neg val_{m-1}$ //PO6
10. **do**
11. $\mathcal{Q} \hat{=} o_i \; ; \; \mathcal{Q} \; ;$
12. $val_{m-1} \hat{=} val_{m-2} \wedge Pre(o_i)$;
13. $val_{m-2} \hat{=} val_{m-2} \wedge \neg Pre(o_i)$;
14. $Opset \hat{=} Opset \setminus \{o_i\}$;
15. **else**
16. **raise exception:** No sequence found;
17. **enddo**
18. **endwhile**
19. **if** $val_0 \Rightarrow val_{m-1}$ **then**
20. $\mathcal{Q} \hat{=} init \; ; \mathcal{Q} \; ;$
21. **else**
22. **raise exception:** No sequence found;
23. **endif**

Figure 3.4: A symbolic search proof based algorithm

3.3.2 Completeness

Although our algorithm succeeds in finding attacks (see Section 3.3.4), it lacks completeness in two cases. First, since operations are deleted from set $Opset$ as soon as they are used, it is not able to extract attack scenarios in which an operation occurs more than once. Second, the algorithm expects operations to have non trivial preconditions, and may stop too early if the sequence includes operations with $true$ as a precondition. In fact, the algorithm builds symbolic states step by step. Based on the precondition of the extracted operation on a given step it infers the previous states as follows:

$$\begin{aligned} val_{m-1} &\hat{=} val_{m-2} \wedge Pre(o_i); \\ val_{m-2} &\hat{=} val_{m-2} \wedge \neg Pre(o_i); \end{aligned}$$

Due to the precondition negation, the algorithm stops when it extracts an operation without precondition (i.e. $Pre(o_i) \hat{=} true$). Consequently, the algorithm may extracts just a part of an existing sequence.

However, experiments shows that it does well for many case studies and it is able to extract symbolic sequences that don't include repetition of operations or operations without precondition or operations with the same preconditions. Because of these restrictions, we propose to combine the symbolic search with model checking. Thus, the symbolic sequence found by our algorithm is then used to guide a model-checker which will instantiate the parameters of operations, and may identify sequences with operation repetition.

3.3.3 Step by Step Illustration

We take advantage of abstraction and step by step we refine the val_{m-2} symbolic state:

1. At the first step of the algorithm, the state space is represented by two symbolic states: the first one val_{m-1} includes all states where the authorization constraint c_m is true and which are enabling op_m, and the second one val_{m-2} is the negation of val_{m-1} which is then $\neg c_m \vee \neg Pre(op_m)$. As they are two disjoint state predicates, we conduct proofs (3) and (6) in order to find an operation o_i that belongs to $\mathcal{O}$ and which is possibly enabled from val_{m-2} and reaches the first state val_{m-1} and such that premise (u, R_u, c_i) is valid. If o_i does not exist, then no sequence could be found for the expected attack and we can try proof (5) for each operation attesting that all operations never reach val_{m-1} from val_{m-2}. Figure 3.5 provides a state machine diagram that illustrates this first iteration. In this representation, states are predicates and transitions are symbolic operations.

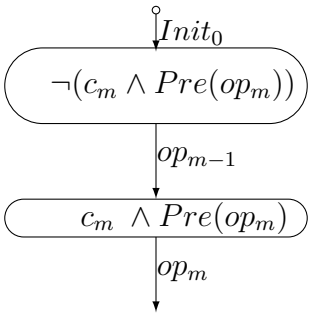

Figure 3.5: First iteration algorithm illustration

2. At the second step of the algorithm, if proofs (3) and (6) succeed for some operation op_{m-1}, then an observable sequence may exist, leading to the critical operation where access control premise (u, R_u, c_m) is valid.

3. If val_0 is inside $Pre(o_i)$ then the algorithm stops. Otherwise, as showed in Figure 3.6, the algorithm looks inside state val_{m-2} in order to find out the previous operations that can be invoked in the attack scenario. State val_{m-2} is partitioned into two sub-states which are:

$$val_{m-2} \wedge Pre(op_{m-1}) \equiv \neg(c_m \wedge Pre(op_m)) \wedge Pre(op_{m-1})$$
$$val_{m-2} \wedge \neg Pre(op_{m-1}) \equiv \neg(c_m \wedge Pre(op_m)) \wedge \neg Pre(op_{m-1})$$

4. Then, we look for operations that reach the first sub-state from the second one.

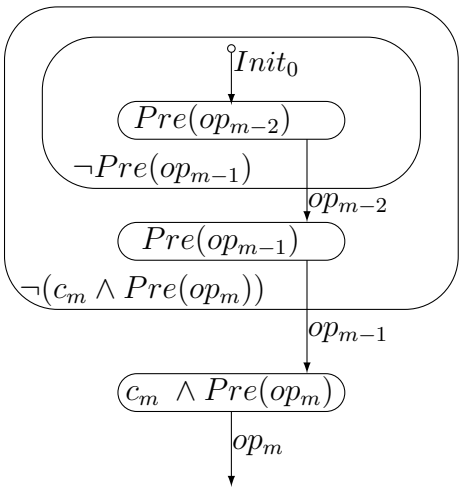

Figure 3.6: Second iteration algorithm illustration

5. The algorithm proceeds iteratively (Figure 3.7) by partitioning the second state into two sub-states until either it finds a state that includes the initial state, which corresponds to a successful search, or fails the search because val_{m-1} is empty or all operations have been invoked once. In the best case, our algorithm gives some symbolic attack scenario, which consists of sequence $(init_0 \ ; \ op_n \ ; \ op_{n+1} \ ; \ \ldots \ ; \ op_m)$ invoked by the same user u and where:

$$val_{n-1} \hat{=} \neg(c_m \wedge Pre(op_m)) \wedge \neg Pre(op_{m-1}) \wedge \neg Pre(op_{m-2}) \wedge \ldots \wedge Pre(op_n)$$

and such that $val_0 \Rightarrow val_{n-1} \wedge \forall i.(i \in (n..m) \Rightarrow (u, R_u, c_i) \models true)$

3.3.4 Application

We apply our algorithm to the meeting scheduler example starting from the following initial state val_0:

$$\begin{aligned} val_0 \hat{=}\ & person = \{Alice, Bob\} \\ & \wedge\ meeting = \{m_1\} \\ & \wedge\ meetingOwner = \{(Alice \mapsto m_1)\} \\ & \wedge\ meetingParticipant = \{(m_1 \mapsto Bob)\} \end{aligned}$$

In this state user John is not allowed to modify meeting m_1 because the authorization constraint allows modification only by the owner of m_1. A malicious scenario would lead

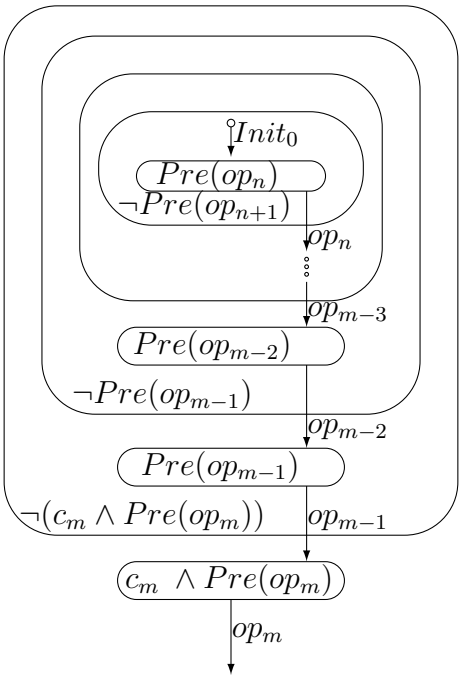

Figure 3.7: Last iteration algorithm illustration

to a state where John becomes able to execute a modification operation such as operation $meetingSetStart$ on meeting m_1. In this state we have to verify:

$$Pre(meetingSetStart(m_1, start)) \hat{=} m_1 \in meeting$$
$$\text{and } (\textit{John}, \textit{SystemUser}, MeetingOwner(m_1) = John) \models true$$

1. First iteration: considering the following symbolic states

$$\begin{aligned} val_{m-1} &\hat{=} (MeetingOwner(m_1) = John) \wedge m_1 \in Meeting \\ val_{m-2} &\hat{=} \neg val_{m-1} \end{aligned}$$

we have:

$val_0 \not\Rightarrow val_{m-1}$ because, in state val_0, $MeetingOwner(m_1) = Alice$,

and

PO (6) is discharged automatically by AtelierB prover for operation $meetingNew$ which may be executed by $John$ as system user, and also for $personAddMeetingOwner$ when $John$ is system administrator. In addition, manual proofs has been done to demonstrate that $meetingNew$ satisfies the PO for instantiation $meetingNew(m_1, John)$ and that $personAddMeetingOwner$ leads to val_{m-1} if and only if it is executed with parameters $(John, m_1)$. We have also checked PO (5) to verify that all other operations never reach val_{m-1} from val_{m-2}. Then, we may go on with the second iteration of the algorithm for each of these two operations.

2. Second iteration: we partition state val_{m-2} into two sub-states:

$$\begin{aligned} val_{m-3} &\hat{=} \quad val_{m-2} \wedge \neg Pre(op_{m-1}) \\ val_{m-2} &\hat{=} \quad val_{m-2} \wedge Pre(op_{m-1}) \end{aligned}$$

- **case 1:** we choose $op_{m-1} = meetingNew(m_1, John)$, and then we have:
  $Pre(meetingNew(m_1, John)) \hat{=} m_1 \notin meeting \wedge John \in person$, and
  $val_0 \not\Rightarrow val_{m-2}$ because $John \notin person$

  In this case, the algorithm does not find an operation permitted to John leading to a state where operation $meetingNew$ becomes enabled. Indeed, no operation satisfies PO (6). In the other side, PO (5) succeeds for all operations. Our algorithm concludes that there does not exist an attack scenario invoking $meetingNew$ in step $m-1$ (Figure 3.8).

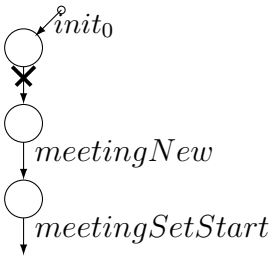

Figure 3.8: No state enabling operation $meetingNew$ is found

- **case 2:** we choose $op_{m-1} = personAddMeetingOwner(John, m_1)$. We have:
  $Pre(personAddMeetingOwner(John, m_1) =$
  $m_1 \in meeting \wedge John \in person \wedge (John, m_1) \notin MeetingOwner$

$$\begin{aligned} val_{m-2} &\hat{=} \neg(m_1 \in meeting \wedge meetingOwner\{(m_1)\} = John) \ \wedge \\ &\quad m_1 \in meeting \wedge John \in person \wedge (John, m_1) \notin MeetingOwner \\ &\hat{=} \quad m_1 \in meeting \wedge John \in person \wedge (John, m_1) \notin MeetingOwner \end{aligned}$$

$$\begin{aligned} val_{m-3} &\hat{=} \neg(m_1 \in meeting \wedge meetingOwner\{(m_1)\} = John) \ \wedge \\ &\quad \neg(m_1 \in meeting \wedge John \in person \wedge (John, m_1) \notin MeetingOwner) \\ &\hat{=} \quad m_1 \notin meeting \vee (meetingOwner\{(m_1)\} \neq John \wedge John \notin person) \end{aligned}$$

  $val_0 \not\Rightarrow val_{m-2}$ because $John \notin person$

  In this case, PO (6) succeeds for operation $personNew$, which means that if this operation is executed, it may lead to a state where operation $meetingNew$ can be enabled. Operation $personNew$ may be executed by $John$ as system administrator.

3. Third iteration: we partition state val_{m-3} into two sub-states:

$$\begin{aligned} val_{m-3} &\hat{=} val_{m-3} \wedge Pre(personNew(John)) \\ &\hat{=} John \notin person \wedge (m_1 \notin meeting \vee meetingOwner\{(m_1)\} \neq John) \end{aligned}$$

This stops normally the algorithm because in this case $val_0 \Rightarrow val_{m-3}$. Figure 3.9 presents the full symbolic scenario that allows John to modify Alice's meeting.

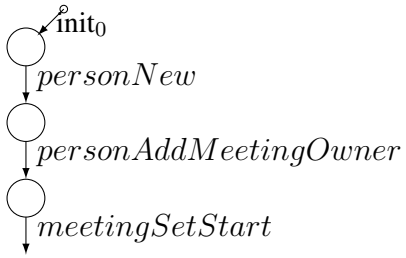

Figure 3.9: Symbolic malicious scenario for user John.

3.3.5 Discussion

Technically, our approach applies the GeneSyst tool in order to produce proof obligations and then asks the AtelierB prover to discharge them automatically. As the resulting scenarios are symbolic and based on "possibly reached proofs", the analyst can conclude that attacks may exist but he can not guarantee their existence for the concrete system. An interesting contribution of our proof-based symbolic sequences, besides the fact that they draw the analyst's attention to potential flaws, is that they give useful inputs to the model-checker. Indeed, a model-checking tool can be used to exhibit, from a symbolic behaviour, an observable concrete sequence of operations that can be followed by an attacker. For example, based on the symbolic sequence of Figure 3.9, ProB was able to extract the malicious concrete scenario represented in Figure 3.10. In order to reduce significantly the state space, we can ask ProB to explore only operations found in the symbolic malicious scenarios. For our example, when trying only operations personNew, personAddMeetingOwner and meetingSetStart, ProB exhibits a concrete attack scenario after visiting a dozen of states which shows a significant speed up with respect to our initial ProB attempts (involving more than 1000 states).

Our technique was able to extract another scenario (Figure 3.11) which can be executed by user Bob holding the supervisor role from the same initial state, in order to steal the ownership of m_1. In this scenario, Bob first cancels the meeting and then he recreates it before applying the critical operation. The first scenario, done by user John, is made possible by the full access permission to class Person, associated to role SystemAdministrator, which includes the right to modify association ends. This attack affects meeting integrity. One solution can be to add a SSD (Static Separation of Duties) constraint between roles SystemAdministrator and SystemUser.

John will then still be able to become owner of the meeting, but will not be able to log in as SystemUser in order to modify it.

The second scenario done by Bob was possible due to role Supervisor which gives him the right to cancel a meeting, and then, as a SystemUser he can recreate it in order to become its owner. This scenario does not point out a flaw since whenever a meeting is cancelled it should be legitimate that a user can start a new meeting with the same identifier as the cancelled one.

Our algorithm was able to extract all possible sequences leading to the state that enables the critical operation. The security breaches that our technique is looking for, are issued from flaws in the conceptual logic of the security policy. Then, the produced sequences don't violate the access control policy, but they must be checked carefully by the security analyst in order to decide whether it is an attack scenario (*e.g* attack 3.10 done by John) or a legitimate access (*e.g* scenario 3.11 performed by Bob).

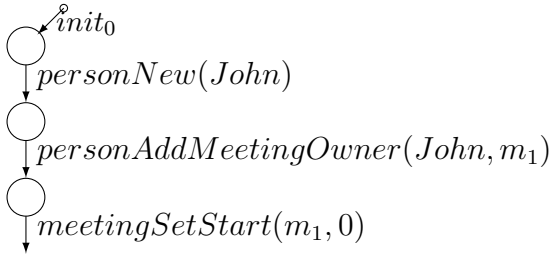

Figure 3.10: John's scenario.

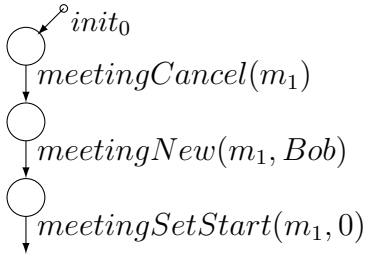

Figure 3.11: Bob's scenario.

3.4 Related Work

Several research works have been devoted to the validation of access control policies. They are mainly focused on detecting external intrusion. Recently, the interest to insider attacks grew leading to two categories of validation: stateless and dynamic access control validation.

Stateless access control validation is dedicated to validate security policies in a given state without taking into account the dynamic evolution of the IS states. Among these works we can

cite the SecureMova tool (Basin et al., 2009) which models security policies using SecureUML and OCL expressions. In the same context, Kuhlmann et al. (2013) took advantage on the USE tool to model RBAC policies in OCL in order to express authorization constraints and to query about access control rules. Far from UML modeling, Fisler et al. (2005) developed the Margrave tool to answer the same kind of stateless queries by analysing RBAC models written in XACML and translated into a form of decision-diagram. Also, some interesting works (Toahchoodee et al., 2009b, Zao et al., 2003) using Alloy language have been defined to analyse access control policies. Even though authors have not sought to study the dynamic evolution of functional states and their effect on the authorization constraints, we believe that a tool like Alloy could support such analysis. Note that such stateless queries can be performed easily based on the formal B specifications produced by the B4MSecure platform. The main limitation of a stateless access control validation is that it does not indicate whether the given state is reachable, or not, and then it may lead to incomplete conclusions.

Dynamic access control validation attempts to identify strategies followed by malicious users to counter some security rules by taking advantage of the IS states evolution. In (Qamar et al., 2011b), we proposed interactive dynamic analysis with the help of a Z animator, but this approach requires user insight and may miss some possible flaws. Zhang et al. (2005, 2008) proposed a backward algorithm to find a strategy allowing the attacker to reach a target state. A plain model-checking approach is proposed in order to check specifications written in the RW (Read-Write) language. On the one hand, the proposed algorithm suffers from scalability for the verification of complex specifications because of combinatorial explosion of the state space; and on the other hand, the RW language is poor compared to B because it doesn't express complex functional behaviours. The authors propose a contribution towards identifying strategies involving multiple users in coalition. This aspect is not covered in this paper and is planned for further work. A similar approach is proposed in (Koleini and Ryan, 2011) in order to validate access control policies of web-based collaborative systems. Even though their experiments show that they achieve better results compared to the approach in (Zhang et al., 2005, 2008), it is still a model-checker solution that can not deal with policies of huge size.

Becker and Nanz (2010) approach is the most similar approach to ours. Indeed, they proposed a Hoare-style proof system based on a Logic for State-Modifying (SMP) in which they express reachability properties in terms of pre and post-conditions. They reason about an abstract set of target states that satisfy some constraint. They also implemented a backward algorithm that extracts the sequence of operations leading to the goal state. Their algorithm is also able to compute the minimal sequence. However, the use of a logic-based language like SMP is not adequate for the context of Information Systems with a need to express complex functional behaviours. Moreover, the use of a proof system on a symbolic model may produce a sequence which cannot be reproduced in the concrete model. To circumvent this shortcoming we proposed in our approach to combine both proofs and model-checking techniques.

Our proof based symbolic search is achieved by assessing reachability properties (Bert et al., 2005) of functional operations. Other works proposed to explore this kind of properties in

order to reason on sequences of B operations. In (Mammar and Frappier, 2015), the authors proposed two approaches to prove reachability properties in a B formal information system modelling. In the first one, they used substitution refinement techniques based on Morgan's specification statement, and in the second one, they proposed an algorithm that produces a proof obligation to be discharged automatically by AtelierB in order to prove whether a given sequence of operations reaches (or not) a defined state. The proposed techniques may help for our work since it is a proposal to simplify proofs and make them easier for AtelierB. However, unlike our approach, they don't search sequences leading to a goal state from an initial one. Their approach starts from a given sequence of operations, and tries to prove its reachability.

3.5 Conclusion

This chapter described a symbolic search approach that can extract insider malicious behaviours from a formal Information System modelling. The meeting scheduler example was discussed in several articles (Basin et al., 2006, 2009). However, they do not report the attack scenarios presented in this paper. This is due to the fact that dynamic evolution of the functional state is not taken into account. We showed how dynamic analysis, assisted by proofs and model-checking, is useful to find out potential threats. In addition, thanks to our algorithm, proofs and model checking tools, our method can be automated in order to extract attack scenarios breaking some authorization constraint. We also applied our approach on several case studies and we were able to find, automatically, the discussed threats.

Our approach is automated thanks to B4MSecure[1], GeneSyst[2], AtelierB[3] and ProB[4]. First, B4MSecure translates functional and security graphical models into B specification, from which we automatically produce proof obligations of enabledness and reachability properties using GeneSyst. Then, these proof obligations are discharged automatically using the AtelierB prover. When a symbolic scenario is found, ProB is used to explore the concrete state space focusing on operations issued from the symbolic scenario. We developed GenISIS, a new tool that implements our algorithm and integrates all the tools mentioned above in order to have a complete automated solution. The main limitation of our work is that sometimes, when proof obligations are complex, AtelierB fails to prove them automatically. Interactive proofs are then required, but they may be pretty difficult for the analyst. One naive solution is to keep operations for which proofs don't succeed automatically in order to be exploited further using the model-checker. A more interesting solution is to focus on other kinds of proof obligations. For example, one can try to prove that an operation op is never enabled from a state E and/or op never reaches a state F. Applying these proofs to the meeting scheduler example we were able to eliminate half of the operations after proving automatically that they cannot be involved in the attack scenario.

[1] http://b4msecure.forge.imag.fr
[2] http://perso.citi.insa-lyon.fr/nstouls/?ZoomSur=GeneSyst
[3] http://www.atelierb.eu/
[4] http://www.stups.uni-duesseldorf.de/ProB

2^{nd} part

Building correct DSLs

Chapter 4

Formal Model-Driven Executable DSLs

Contents

This chapter is an excerpt issued from these two papers:

Akram Idani. Formal model-driven executable DSLs: Application to Petri-nets. *International Journal on Innovations in Systems and Software Engineering (ISSE)*, 18(4), 2022. URL https://doi.org/10.1007/s11334-021-00408-4.

Akram Idani, Yves Ledru, and German Vega. Alliance of Model Driven Engineering with a Proof-based Formal Approach. *International Journal on Innovations in Systems and Software Engineering (ISSE)*, 16(3):289–307, 2020. URL https://doi.org/10.1007/s11334-020-00366-3.

Model Driven Engineering (MDE) is an interesting paradigm in software systems development because it provides solutions to the software complexity on the one hand, and it shows how to bridge the gap between conceptual models and coding

activities, on the other hand. The definition and the use of domain specific models throughout the engineering life-cycle makes MDE a powerful asset. Furthermore, MDE is assisted by numerous tools (EMF[1], XText[2], ATL[3], etc) dedicated to put into practice a clear separation of concerns ranging from requirements to target platforms, and going through several design stages. Interoperability between these tools is favored by the use of standardized meta-modeling formalisms which increases automation especially for developing domain specific modeling languages (called DSMLs). These advantages reduce the risk that human errors such as misinterpretation of the requirements and specification documents lead to erroneously validate the specification, and hence to produce the wrong system. Still, while model-driven DSLs provide solutions to the validation problem ("do the right system"), the verification problem ("do the system right") remains a major challenge due to the lack of formal reasoning tools in MDE platforms.

Contribution

This chapter shows how we define mappings between DS(M)Ls as they are defined in MDE techniques and the rigorous world of formal methods. Our approach is assisted by Meeduse (Idani, A., 2020b), a tool that we developed in order to translate a DS(M)L meta-model into an equivalent formal B specification that represents its semantic domain, using the set theory and the first order predicate logic. In MDE, the design of a DS(M)L addresses two main layers: meta-modeling and modeling. Meta-modeling refers to the definition of the language abstract syntax where domain concepts and their relationships are defined independently from their concrete representation. The modeling layer allows one to create instances of these domain concepts by applying a predefined concrete syntax (graphical or textual). Our approach applies the formal B language to both layers. At the modeling layer, the abstract data-types generated from the meta-model layer, become valuated data types, which allows animation. Regarding the DSL behavioral semantics, they follow the same principles. Indeed, they are defined over the meta-modeling layer using high-level B operations, which are then animated by Meeduse on the modeling layer.

This alliance between MDE and a formal method, assisted by Meeduse, makes domain specific models provable and also executable thanks to the animation of their expected behaviour directly in a dedicated DS(M)L tool. The interest of Meeduse is that it integrates the ProB JAVA API (Körner et al., 2020). Given a model designed in the DS(M)L tool, Meeduse injects it as valuations in the generated B specification and calls ProB in order to compute the list of operations that may be animated from these valuations. When an operation is animated the tool computes the new variable valuations and then it translates back these valuations to the initial model which results in an automatic animation of domain models. This technique allows one to

[1] EMF: https://www.eclipse.org/modeling/emf/
[2] Xtext: https://www.eclipse.org/Xtext/
[3] ATL: http://www.eclipse.org/atl/

take benefit of formal reasoning tools such as provers and model-checkers throughout the MDE process. The proposed method uses the AtelierB prover to guarantee the correctness of the model's behavior with respect to its invariant properties, and the ProB model-checker in order to animate underlying execution scenarios. Besides the use of these automatic reasoning tools in MDE, proved B refinements have been investigated in order to gradually translate abstract models to concrete ones that can then be automatically compiled into a programming language.

Structure

This chapter is structured as follows:

- Section 4.1 discusses the meaning of DS(M)Ls semantics and presents an overview of our approach to formally define these semantics.
- Section 4.2 summarizes an experimental study that we have done with several tools dedicated to executable DSLs, and gives our observations and lessons learned.
- Section 4.3 shows how the semantics of a DSL can be formally defined using the B method and CSP.
- Section 4.4 illustrates some debugging activities that can be done using the Meeduse tool.
- Section 4.5 draws the conclusion of this chapter.

4.1 DS(M)L's semantics

Execution of Domain Specific (Modeling) Languages (DS(M)Ls) is an active research area in Model Driven Engineering (MDE). The intention is to be able to perform early analysis of a system's behavior before its implementation. Indeed, execution a DS(M)L tends to reduce the gap between the model and the system, since DS(M)Ls would not only represent the expected system's structure and behaviour but they can themselves behave as the system should run. In this work we apply a formal approach in order to define, prove, and execute the underlying semantics of a DS(M)L. This section presents an overview about DS(M)L semantics as they are defined in the state of the art and explains the main principles of our approach.

4.1.1 Abstract syntax and semantic domains

Bryant et al. (2011) discuss challenges and directions about DSLs semantics and point out that their formal definition is the foundation for several benefits: automatic generation of some DSL tooling, formal analysis of a model's behavior, and/or composition of concerns issued from different languages. The authors define the semantics of a DSL as a mapping from its abstract syntax to some semantic domain. The abstract syntax defines the modeling concepts, their

relationships and attributes; and the semantic domain is some mathematical framework whose meaning is well-defined. In the same direction, Harel and Rumpe (2004) state that a semantic domain is a well-defined and well-understood agreement on a language's meaning.

In MDE, the notion of abstract syntax is well mastered today thanks to the concept of meta-model (which is standardized by the Meta-Object Facility (MOF) document (Object Management Group, 2015)). However, the notion of semantic domain remains unclear because, on the one hand, the MOF is informally defined, and on the other hand, a common misconception in modeling languages is to confuse semantics with behaviours as stated in (Harel and Rumpe, 2004). In fact, the description of a behaviour is itself a DSL with its own abstract syntax and semantic domain. For example, several MDE tools define the simple Petri-net DSL using a common meta-model, but they apply various languages (Kermeta, xMof, fUML, java) to define the expected behaviour when firing a petri-net transition. These languages are additional DSLs with different semantic domains that lead to different behaviours when executing the same Petri-net model with different tools.

Meta-models are therefore not sufficient to describe the behaviour of a DSL, that's why some attempts exist in order to enhance meta-models by action languages, *e.g.* Kermeta (Bousse et al., 2018); or to translate meta-models into a target language that offers behavioural facilities, *e.g.* Java (Hartmann and Sadilek, 2008). In our approach we apply this second strategy, called translational in (Bryant et al., 2011), but we use the formal B language (Abrial, 1996), which offers a way to mathematically define the semantic domain of a DSL. In our approach we adopt a deep embedding approach when translating a DSL meta-model into the B language. The formal semantic domain of a DSL will represent data typing with the corresponding structural constraints defined in set theory and first order logic. Regarding the DSL behaviour, it is defined using B operations that can be manually written, like presented in this chapter or generated from additional DSLs that can describe behaviours. In (Idani, A., 2020b) we proposed to produce the B operations from a coloured Petri-nets DSL.

4.1.2 Dynamic semantics vs Behavioural semantics

Several research works have been devoted in order to provide solutions with tools that make DSLs executable. In Model-Driven Engineering, DSL execution mainly refers to the definition of semantics, which are roughly called by means of several terms like dynamic, operational, executable and/or behavioural. Jézéquel et al. (2013) refer to behavioral semantics of a DSL, as the semantics that allow to run a program. They identify two kinds of behavioral semantics: translational semantics and operational semantics: a translational semantics would result in a compiler while an operational semantics would result in an interpreter. In (Muller et al., 2005), a different definition is provided without a distinction between operational semantics and behavioral semantics. The authors refer to a high level execution defined over meta-models and which uses actions (also called operations) to define model's behaviour. Similarly to programs which are made of data structures and algorithms, in (Muller et al., 2005) an executable DSL is made of meta-data and actions. The various definitions that we found mix low-level notions

(compilers, programs, etc), and high level notions (meta-models, etc). This is less the case for programming languages where the terms dynamic semantics and execution semantics are used to describe the runtime behavior of programs (Vergu et al., 2015). The so-called dynamic (or execution) semantics in programming languages, "*defines how and when the various constructs of the language produce artifacts* (*e.g.* bytecode) *that can be executed in a given target platform* (Floyd, 1993) (*e.g.* virtual machine)".

For clarity we use terms dynamic and execution semantics of a DSL as soon as a runtime environment is concerned which is close to programming languages (Vergu et al., 2015, Floyd, 1993). Terms operational and behavioral semantics for us are inspired by (Muller et al., 2005) and they are limited to high level actions that describe how models behave. The gap between both semantics, does not make them conflicting because at a high level one can define behaviours of models and at a low level this behaviour can be translated into a runtime environment. The challenge is how to guarantee the compatibility between the resulting executions: that of a model given its operational semantics, and that of the runtime program given its dynamic semantics. Works of U. Tikhonova (Tikhonova et al., 2013, Tikhonova, 2017a,b) bridge this gap between operational and dynamic semantics, by providing reusable specification templates which are templates that allow to explicitly define the dynamic semantics of a DSL over its operational semantics. These templates are presented in the form of a library of specifications, each of which formalizes a separate software solution. The operational semantics of a DSL are then expressed as a composition of these specification templates, which favours some kind of generic programming of the DSL dynamic semantics. However, the use of a specification templates library makes several restrictions to the definition of semantics because it is not obvious to cover all possible behavioural needs with various abstraction levels. In our proposal, we address the semantic gap between operational and dynamic semantics of a DSL thanks to the refinement paradigm of the B method which allows to produce step-by-step a low level implementation of a DSL from its high level description.

4.1.3 Executable DS(M)Ls in Meeduse

Figure 4.1 summarizes how we put into practice the alliance of MDE with a proof-based formal approach in order to define DS(M)Ls with formal semantics and favour their execution. Our tool support named Meeduse[4] links three technological spaces: EMF (Steinberg et al., 2009) for model driven engineering, B Method (Abrial, 1996) for proofs and refinements, and finally the execution of the target system. The top hand side of figure 4.1 represents the abstract syntax of a DSL by means of an ECore meta-model, its mapping to the corresponding formal semantics domain (box B Machine), and its operational semantics which starts from the description of an abstract behavior and applies refinements until the specification of its concrete behavior (boxes abstract behavior, refined behavior and concrete behavior). The bottom hand side of figure 4.1 defines the dynamic semantics of the DSL since it is an extraction of an executable language

[4] Meeduse: Modeling Efficiently EnD USEr needs (`http://vasco.imag.fr/tools/meeduse/`).

from the high level DSL artifacts. We build our approach on several components: Translator, Injector, Animator and Trace runner.

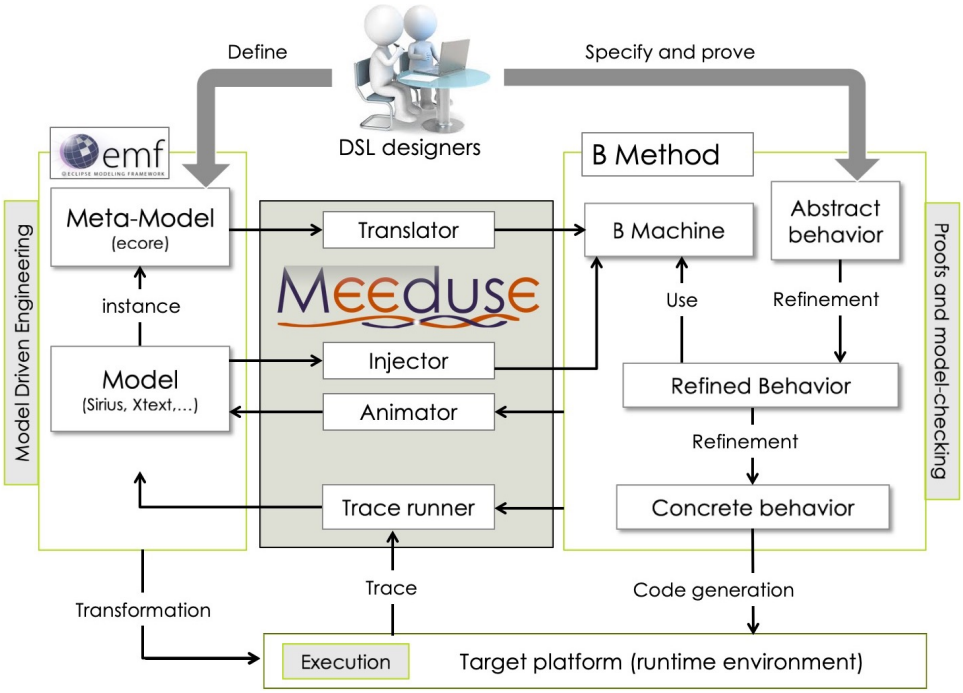

Figure 4.1: Meeduse overview

(1) Translator: this component translates an Ecore (Steinberg et al., 2009) meta-model into an equivalent B specification gathering the structure of the meta-model as well as basic operations like constructors, destructors, getters and setters. The resulting B specification can be enhanced manually by additional invariants and its proof of correctness with respect to these invariants can be done using AtelierB prover. Technically, the tool translates an ECore model into an UML model and then applies B4MSecure to produce a formal B machine representing the functional aspects of the DSL.

(2) Injector: injects instances of a meta-model, which can be designed using EMF-based modeling tools (like Sirius, GMF, XText, etc) into the B specification produced from the meta-model. This component introduces enumerations into abstract data structures like abstract sets, and hence allows valuations of the B machine variables.

(3) Animator: animation of B specifications is done using the ProB JAVA API (Körner et al., 2020). The Animator component asks ProB to animate B operations and gets the reached state by means of B variables valuations. Then, Meeduse computes the equivalence between these valuations and the initial EMF model and applies the necessary modifications to the model in order to keep this equivalence all along the animation process.

(4) Trace runner: this component plays a sequence of operations issued from an execution trace by animating the corresponding B operations, which leads to automatic modifications of the model. Thanks to this component, animation can be done from outside EMF by an external program which is running in the target platform. Note that the target platform source code can be produced by applying MDE transformations or by using the code generator of AtelierB after successive refinements. In both cases, the trace runner is useful on the one hand, for conformance validation between the model and the execution, and on the other hand for some kind of runtime verification. In (Idani, A. et al., 2020), we presented the first strategy where an implementation in the C language is generated by AtelierB from the most concrete B machine.

4.2 Preliminary study and critical review

In (Idani, A., 2022, 2020a) we experimented several existing MDE implementations of Petri-nets applying Java, QVT, Kermeta and fUML. We tried them to debug a safety-critical system in order to check their ability to address properties such as: correctness, deadlock freedom, mutual exclusion and fairness. This section gives the lessons learned from this study and discusses why formal alternatives are required to ensure the correctness of DSLs.

4.2.1 A simplified Petri-net DSL

Petri-net (Petri and Reisig, 2008) is a visual language used for modelling concurrent systems. Its mathematical foundations inspired by the graph theory allow formal calculus about safety properties. The choice of this DSL is motivated by the fact that it is often used as an illustrative case by the research works and tools interested in modeling and debugging techniques, and also because it has had a wide range of applications in safety critical systems. This section builds on a simplified version of this DSL and defines a Petri-net based safety-critical example.

4.2.1.1 Structural and contextual semantics

Figure 4.2 shows the simplified Petri-net meta-model as considered by (Bousse et al., 2018, Deantoni, 2016). It is composed of three meta-classes: Net (the root class), Place and Transition. These classes are linked by four relationships: places, transitions, input and output. This meta-model defines structural features of a given Petri-net. For instance, a transition must be linked to at least one input place and one output place. Attribute tokens represents the number of tokens in a place: it is single-valuated, optional and without a default initial value. The various references of this meta-model don't admit repetitions. Note that the meta-model is taken from (Bousse et al., 2018)[5] and it is presented without any modification. There exist some variations of this meta-model where transitions admit both 0 inputs and 0 outputs. Furthermore, the Petri-net DSL must comply with the following contextual invariant written in OCL:

[5] The corresponding ECore file can be found at (Petri-net ecore file, 2020).

```
context Place
inv Token_Is_Natural:  self.tokens ≥ 0
```

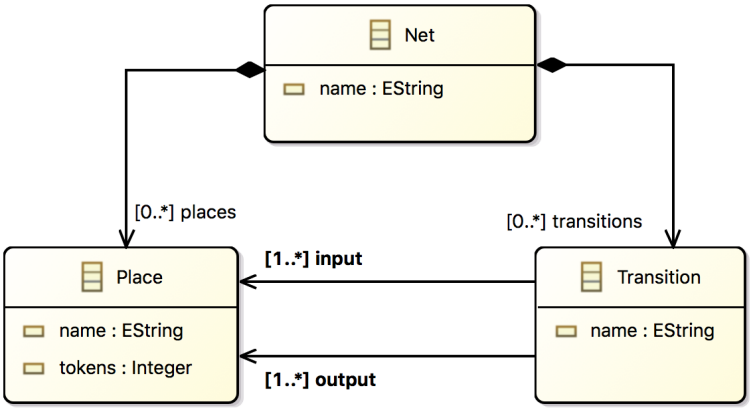

Figure 4.2: Petri-nets meta-model.

For illustration we use the Petri-net of figure 4.3 that controls traffic lights in a crossroad. In some sense, this model deals with a safety-critical system since failures may lead to loss of life due to accidents that it may cause. This model represents two traffic lights (Light 1 and Light 2) that are to be placed in two roads that intersect. These traffic lights are respectively controlled by the left side and the right side of this figure. Every traffic light sequentially switches from Green to Orange and then to Red, in an infinite loop. This Petri-net model shows concurrent evolutions of traffic lights without any synchronisation between them. Finally, the current state of this model assigns red to Light 1 and green to Light 2.

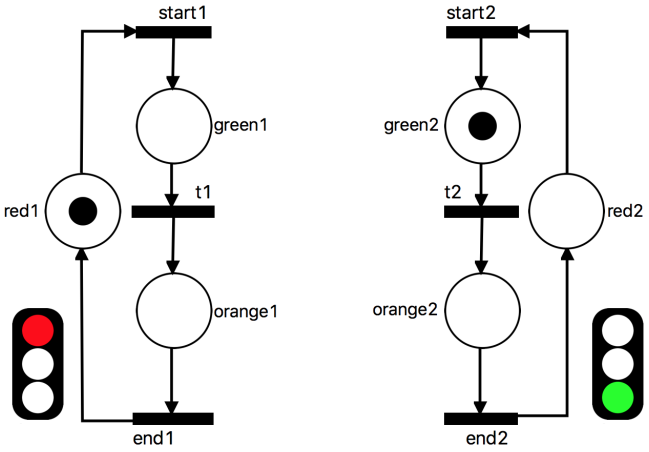

Figure 4.3: Traffic light controller in petri-nets (V1)

In our study we supposed that by using a Petri-net debugger, a domain expert would like to verify some safety properties such as:

- correctness: asserts that the system does not exhibit bad behaviors, where invariants (structural or contextual) are violated.
- deadlock-freedom: states that the traffic lights can't be blocked in a state in which no progress is possible
- mutual exclusion: states that lights in a road intersection cannot enter simultaneously their critical sections (critical sections are states green and orange in our example).
- fairness: requires that the system gives fair turns to its components (in our example both lights must be able to function).

4.2.1.2 Execution semantics

Basic execution semantics of the Petri-net DSL are defined by means of transition firing which holds when a transition satisfies an enabledness property. To check this property, we require to call a query defined as:

```
query isEnabled(t : Transition) : Boolean =
    t.input->forAll(p : Place | p.tokens > 0)
```

This query returns true if attribute tokens is greater than 0 for each input place of transition t, false otherwise. Algorithm of figure 4.4, taken from (Bousse et al., 2018), describes how a Petri-net runs.

Algorithm 1: run

Input:
n : the Net object to run

```
[1] begin
[2]     t_enabled :∈ {t ∈ n.transitions | isEnabled(t)}
[3]     while t_enabled ≠ null do
[4]         fire(t_enabled)
[5]         t_enabled :∈ {t ∈ n.transitions | isEnabled(t)}
```

Algorithm 2: fire

Input:
t : the Transition object to fire

```
[1] begin
[2]     foreach p ∈ t.input do
[3]         removeToken(p)
[4]     foreach p ∈ t.output do
[5]         addToken(p)
```

Figure 4.4: Running a Petri-net.

This algorithm chooses non-deterministically (operator $:\in$) a transition t (called $t_{enabled}$) from the set of transitions that satisfy the enabledness property and then calls operation $fire(t)$.

As a result the number of tokens in the input places of t is decreased (operation $removeToken$) and the number of tokens in the output places is increased (operation $addToken$). Modifications of tokens, done at every call to operation fire, modify the set of enabled transitions. The algorithm may loop or stop when this set becomes empty.

4.2.1.3 Benchmark

In order to debug the aforementioned safety properties using various MDE approaches and observe their strengths and limitations, our experimental study was built on approaches that are widely accepted in the MDE literature: Java, QVT, Kermeta and fUML. In the remainder, we refer to them respectively as: $\text{PNet}_{\text{Java}}$, PNet_{QVT}, $\text{PNet}_{\text{Kermeta}}$ and $\text{PNet}_{\text{fUML}}$. Our choice was also motivated by the fact that these approaches are the only ones that we found, where the Petri-net DSL was already implemented, in addition to the availability of the underlying artifacts (source code, publications, tutorials). Furthermore, tools assisting these approaches use the Eclipse Modeling Framework (EMF), which makes easy their integration and the analysis of the Petri-net DSL within a unified MDE framework.

$\text{PNet}_{\text{Java}}$ (Hartmann and Sadilek, 2008). Java-based semantics of the petri-net DSL are proposed in (Hartmann and Sadilek, 2008) and supported by a tool named EProvide (2020). This java implementation is easily reproducible in the Eclipse Modeling Framework. The approach is combined with graphical editor creation in order to support rapid prototyping of animated visual interpreters and debuggers.

PNet_{QVT} (Wachsmuth, 2008). QVT (Query/View/Transformation) is an OMG standard for model transformations. QVT defines: QVT-Relations and QVT-Core which are declarative languages but at two different levels of abstraction, and QVT-Operational which is an imperative language. Wachsmuth (2008) used QVT-Relations which is the high-level language of QVT extending OCL and its semantics with imperative features.

$\text{PNet}_{\text{Kermeta}}$ (Deantoni, 2016, Bousse et al., 2018). Kermeta (Jézéquel et al., 2013) is a language workbench that involves different meta-languages for abstract syntax (aligned with EMOF (Object Management Group, 2015)), static semantics (aligned with OCL) and behavioural semantics. In (Deantoni, 2016, Bousse et al., 2018), the Gemoc studio was applied together with the Kermeta/XTend language to define the petri-net DSL and debug its execution using an animation technique.

$\text{PNet}_{\text{fUML}}$ (Langer et al., 2014, Mayerhofer et al., 2013). fUML (Object Management Group, 2011) is an OMG standard that defines the execution semantics of a subset of UML 2.3. The standard applies, in the form of pseudo Java-code, a basic virtual machine enabling UML models using elements comprised in the fUML subset to be executed. Mayerhofer et al. (2013)

proposes the xMOF tool which integrates fUML with MOF to enable the specification of the behavioural semantics of DSLs in terms of fUML activities.

4.2.2 Ascertainment and discussion

4.2.2.1 Challenges

There exist several well established tools (Thong and Ameedeen, 2015) that implement the Petri-net theory for verification purposes and apply various programming languages. However, even if most of them are open-source, it remains difficult to update their code in order to experiment various semantics. Not only the developer must have high programming skills, but he/she must also understand the tool logic, which is a very challenging task especially for tools dedicated to formal languages. To this purpose, a MDE solution is much more suitable, because it allows one to reason on the DSL itself, rather than on how the DSL is encoded in a given programming language. Our works focus on MDE tools dedicated to DSL execution and debugging, and apply them to the Petri-net DSL, together with a formal approach. The objective is to benefit from the rich catalog of MDE tools without losing sight of correctness and rigorous development. Indeed several MDE tools exist: model-to-model transformation, model-to-code generation, constraint-checkers, graphical concrete syntax representation, bi-directional mappings, etc. The approach of $\text{PNet}_{\text{Java}}$ proposes to support stepping back in the execution history of a DSL. PNet_{QVT} investigated model-to-model transformations as a way to define execution semantics of a DSL. $\text{PNet}_{\text{Kermeta}}$ relies on generic trace management to provide an omniscient debugger thereby allowing developers to "go back in time" (Lienhard et al., 2008). Finally, $\text{PNet}_{\text{fUML}}$ proposed a new meta-modeling language (xMOF) for specifying the abstract syntax and behavioral semantics of DSLs based exclusively on standardized modeling languages (fUML and MOF). We experimented these tools being guided by two point of views:

- The end-user point of view: the end-user of a Petri-net is often either a Petri-net expert who is interested by verification features (such as model-checking), or a domain expert who is interested by interactive animation and/or simulation. Note that the experimented approaches were not interested by verification, but rather by validation; and hence we focused on debugging rather than on proofs and model-checking.

- The developer point of view: we assume that the developer of a Petri-net DSL has a good knowledge of the Petri-net theory and his/her intention is to provide a bug-free tool. Thus, we look at how existing tools encoded the semantics of the Petri-net DSL in order to locate the critical parts and find the origin of failures when they happen.

4.2.2.2 Observations

Unfortunately, even if our benchmark tools address several interesting features and challenging directions for DSL development, their application to the Petri-net DSL is limited to the

simplified semantics. As they didn't go further towards the realistic Petri-net DSL, it is difficult to objectively evaluate the usefulness of the resulting models for verification. Hence, our objective was not to check the correctness of their Petri-net case study − since their underlying approaches didn't deal with correctness − but rather to build a formal MDE Petri-net and illustrate its contributions based on existing simplified implementations from which unsafe behaviours were exhibited. Indeed, our main observation is that even if it seems easy to use MDE tools to develop the semantics of a language, the developer can miss obvious details without a good support for verification. Furthermore, this may weaken the general approach of executable DSLs when applied for debugging complex and safety-critical systems.

The analysis of our benchmark for simplified Petri-nets showed that failures or unsafe behaviours may originate from several artifacts:

- The meta-model and its underlying modeling operations, because often execution semantics apply the modeling operations to update the input model. In order to safely update the model, our approach generates a set of pre-established modeling operations that are correct by construction, meaning that they will never violate the structural features of the meta-model.

- The model itself, because incorrect behaviours may not be exhibited from models, especially when the DSL builder applies internal choices that are not conformant to the standard semantics. To cope with this issue we have done a systematic analysis of the MOF semantics for all the constructs that are covered by our approach and provided a suitable transformation from ECore to B.

- The implementation of the operational semantics that may be distant or not conformant to the DSL's specification. This is a validation problem that requires the involvement of a domain expert. In the context of the Petri-net DSL the semantics are mathematically defined, which is compatible with the usage of a formal method such as B.

- The execution engine of the meta-language. For example, EMF/OCL based engines wrongly support non-determinism as discussed in (Baar, 2005, Vallecillo and Gogolla, 2017). In our case, the execution engine is the ProB animator and model-checker (Leuschel and Butler, 2008). This powerful tool has had several successful industrial applications that address safety-critical systems.

The four implementations of our benchmark tools are highly simplified and used for illustration only. The underlying approaches are accepted by the MDE community and have the ability to design and correctly instrument DSLs. From our experiments $\text{PNet}_{\text{Java}}$ gave the better execution outputs however it suffers from its poor abstraction. PNet_{QVT} gave the better abstraction level however it suffers from limitations of the misuse of non-determinism. Finally, $\text{PNet}_{\text{Kermeta}}$ and $\text{PNet}_{\text{fUML}}$ proposed a controllable deterministic behaviour by enabling the first transition satisfying the enabledness property.

4.2.2.3 Added-value and limitations

To cope with the aforementioned limitations, we proposed an alternative solution, to the Petri-net DSL. The correctness of our solution is attested by theorem proving and its execution is managed by an animator. Section 4.3 presents our solution and Section 4.4 shows how the B method can help during the verification and validation activities. The challenge is therefore to show by practice how and why a formal model-driven executable DSL is developed and executed. The Petri-net DSL is widely applied in safety-critical systems for verification and simulation (Thong and Ameedeen, 2015), and hence we believe that the application of a formal method to this DSL is a strong requirement. Moreover, in order to go beyond the simplified illustrative case we applied our solution to PNML (Petri-Net Markup Language), the international standard ISO/IEC 15909 for Petri-nets, providing formal and realistic basis that may be useful for possible improvements of other MDE tools. This application is presented in the next chapter.

A major limitation of our FMDE approach is that the integration of a formal method within MDE tools is not an easy task. Indeed, a major difficulty is that usually a MDE expert does not have knowledge of formal methods; and often, proofs and model-checking is not widespread because it seems to create an overhead for the developer. In (Andova et al., 2012), the authors state that: "*[...] the learning curve of formal methods is steep, whereas the learning curve for drawing diagrams on the black board is very low.*". This observation is true due to the complexity of the mathematical notations that support formal methods.

Nonetheless, when language analysers, such as compilers, are used for safety-critical or high-assurance software, Leroy (2009) attests that "*validation by testing reaches its limits and needs to be complemented or even replaced by the use of formal methods such as model checking, static analysis, and program proof*". In our proposal we advocate for collaborations between the formal methods community and the MDE community in order to take benefits of their respective tooling. Our approach (Idani, A. et al., 2020), and its tool support Meeduse (Meeduse, 2020, Idani, A., 2020b), favor this collaboration since it makes possible the use of DSL builders and formal methods tools together in one unified framework (*i.e.* EMF). The DSL tool development is the task of MDE experts who have the skills to define meta-models with associated static constraints, and the formal semantics specification is the task of the formal methods experts who are experienced in provers and model checkers.

4.3 A Formal Model-Driven Petri-net DSL

The disparity between DSL execution tools leads to behaviours that are conformant to the semantics specified by their execution models but which may be far from the expected behaviour in accordance with the domain expertise. This is an important problem since the same model may not be executed in the same way on different tools even for deterministic structures. In fact, when designing a model via a given DSL tool, the domain expert focuses on debugging his/her model rather than debugging the DSL semantics provided by the MDE expert.

In (Idani, A., 2022), we proposed an alternative definition of the Petri-net semantics (that we call $\text{PNet}_{\text{Reference}}$) using Meeduse. The use of a well-established formal approach assisted by provers and model-checkers, would at least guarantee the consistency of the Petri-net DSL and its conformance to the expected semantics. The proposed formal model-driven Petri-net DSL builds on the B method (Abrial, 1996) and the CSP (Hoare, 1985) process algebra (Communicating Sequential Processes).

4.3.1 Static semantics

In order to get a formal B specification of the static semantics of a DSL, Meeduse translates its meta-model into a correct-by-design B specification. This translation extracts a UML model from the ECore file of the meta-model, and then applies B4MSecure. Figure 4.5 gives the structural part of the generated B machine.

```
1.  MACHINE
2.      nets
3.  SETS
4.      NET; PLACE; TRANSITION
5.  ABSTRACT_VARIABLES
6.      Net, Place, Transition,
7.      places, input,
8.      output, transitions, Place_tokens
```

```
9.  INVARIANT
10.     Net ∈ F (NET)
11.     ∧ Place ∈ F (PLACE)
12.     ∧ Transition ∈ F (TRANSITION)
13.     ∧ places ∈ Place ⇸ Net
14.     ∧ input ∈ Place ↔ Transition
15.     ∧ output ∈ Place ↔ Transition
16.     ∧ transitions ∈ Transition ⇸ Net
17.     ∧ Place_tokens ∈ Place ⇸ NAT
18.     ∧ ran(input) = Transition
19.     ∧ ran(output) = Transition
20.     ∧ ∀ transition · (transition ∈ ran(input)
21.         ⇒ input ⁻¹ [{transition}] ≠ ∅ )
22.     ∧ ∀ transition · (transition ∈ ran(output)
23.         ⇒ output ⁻¹ [{transition}] ≠ ∅ )
```

Figure 4.5: Structural part of the Petri-nets machine

The invariant covers structural properties defined by multiplicities and the optional/mandatory character of attributes, as well as contextual constraints like the `Token_Is_Natural` invariant. For example, predicates from line `(18.)` to line `(23.)` translate multiplicities `1..*` associated to references input and output. Attribute tokens which is single-valued, optional and defined over the set of natural numbers, is translated into a partial function from set Place to the B type NAT (see line `(17.)`). Note that in EMF this type does not exist, which then requires the additional OCL constraint in the EMF-based tools. In Meeduse it is possible to mix predefined EMF/UML basic types with those defined in the B language.

4.3.2 Modeling operations

Our objective is to use Meeduse in order to reproduce the EMF process for editing models, but in the rigorous world of a formal method. Indeed, EMF generates a Java implementation from

a meta-model gathering all basic operations (setters, getters, etc) and Meeduse generates a B machine gathering similar basic operations but which are written in a theory (set theory, first order predicate logic and generalized substitutions) allowing to carry out proof of correctness. Figure 4.6 shows the basic setter of attribute `tokens` produced by Meeduse. By default, this setter preserves invariant properties since it affects a natural number to attribute `tokens` thanks to precondition: $val \in \mathsf{NAT}$.

Place_SetTokens($aPlace$, val) =
PRE
  $aPlace \in Place \wedge val \in \mathbf{NAT}$
THEN
  $Place\_tokens :=$
    $(\{aPlace\} \lhd Place\_tokens) \cup \{(aPlace \mapsto val)\}$
END

Figure 4.6: Basic setter of attribute tokens

Figure 4.7 gives two cases of the place creation operation that depend on the optional/mandatory character of attribute tokens. An attribute without an initial value in the meta-model leads to two possibilities: (1) if the attribute is mandatory then the B constructor requires a parameter to assign an initial value to the attribute and (2) if it is optional then the constructor creates a place with an undefined attribute value.

Mandatory $Place \rightarrow \mathbf{NAT}$	Optional $Place \nrightarrow \mathbf{NAT}$
Place_NEW($aPlace$, val) = **PRE** $aPlace \in PLACE \wedge$ $val \in \mathbf{NAT} \wedge$ $aPlace \notin Place$ **THEN** $Place := Place \cup \{aPlace\} \parallel$ $Place\_tokens :=$ $Place\_tokens \cup \{(aPlace \mapsto val)\}$ **END**;	**Place_NEW**($aPlace$) = **PRE** $aPlace \in PLACE \wedge$ $aPlace \notin Place$ **THEN** $Place := Place \cup \{aPlace\}$ **END**;

Figure 4.7: Constructors of class Place

The B operations generated by Meeduse are conceptually more accurate than the unique java translation used in EMF-based DSL tools because they guarantee the preservation of the structural features of the meta-model. From this specification, AtelierB generated 74 proof obligations (POs) for which it was able to automatically prove 62. The 12 other POs were proved interactively without any improvement of the B specifications.

4.3.3 Operational semantics

Tools of our benchmark used (meta-)programming languages with algorithmic facilities in order to define the operational semantics of Petri-nets. In our approach we use additional B operations to those generated automatically and we propose to coordinate them using CSP process algebra. Besides theorem proving, this formal background, allows one to apply associated animation and model-checking tools in order to rigorously debug and verify the various models.

Operational semantics often introduce complex modifications to the domain-specific model: they may create or destroy objects, modify relationships between these objects and also update several class attributes at the same time. We are then afraid that the difficulty in applying executable DSLs in safety-critical systems goes beyond the failures we were able to exhibit from our benchmark tools, *i.e.* overflow typing or implicit assignment of default values, or even non-determinism. In our opinion, this issue needs a methodological background and a clear separation of concerns regarding the properties to verify:

1. those of the meta-model with associated modeling operations (*e.g.* setToken,...),
2. those of the user-defined operations (*e.g.* addToken, removeToken,...),
3. those of the coordination mechanism (*e.g.* operations fire and run of figure 4.4).

The first kind of properties is covered by the functional model (machine nets of figure 4.5) extracted automatically from the meta-model. The resulting B specification gathers modeling operations and can be enhanced by additional invariants in order to take into account contextual properties and prove its correctness. The two other kinds of properties would be introduced by an expert in formal methods based on this functional model. This is a similar approach to a classical MDE technique where first EMF generates a Java API from a meta-model and then frameworks for executable semantics suggest the use of specific languages (like QVT, Kermeta, etc). In our approach this specific language is that of the B method.

We define operational semantics using B operations in an additional machine that we call semantics in the following. This machine includes the functional machine nets in order to be able to explicitly call its operations. Contrary to the java protected field produced by EMF from attribute tokens, in B the only way to modify a variable outside the machine in which it is defined, is to use operations provided by the latter. The Petri-net implementations of our benchmark are simple cases because they should address mainly two basic properties: the first one about attribute tokens that belongs to natural numbers was discussed in the previous section, and the second one is about transition enabledness. This property should not only take into account the positive value of tokens (relation Place_tokens) for all input places ($\mathsf{input}^{-1}[\{\mathsf{tt}\}]$) of a transition tt but must also take into account the upper limit of this attribute for all output places ($\mathsf{output}^{-1}[\{\mathsf{tt}\}]$). Operation getEnabled (figure 4.8) is a formalisation of query isEnabled presented in section 4.2.1.2. Substitution ANY ... WHERE ... END gets any transition tt satisfying preconditions:

(P1) Place_Tokens[input^{-1}[{tt}]] $\cap$ $\{0\} = \emptyset$
(P2) Place_Tokens[output^{-1}[{t}]] $\cap$ $\{$MAXINT$\} = \emptyset$

$tEnabled \leftarrow$ **getEnabled** $=$
ANY tt **WHERE**
  $tt \in Transition \wedge$
  $\{0\} \cap Place\_tokens[input^{-1}\ [\{tt\}]] = \emptyset \wedge$
  $\{\mathbf{MAXINT}\} \cap Place\_tokens[output^{-1}\ [\{tt\}]] = \emptyset$
THEN
  $tEnabled := tt$
END;

Figure 4.8: Operation getEnabled

Since relation Place_tokens is defined over natural numbers, it is not necessary to use the forAll primitive inside the precondition like in the OCL query. The precondition of operation getEnabled simply verifies if values 0 and MAXINT belong to the sets of tokens issued from the input and output places of tt. Precondition (P1) is not sufficient because we would like to safely increase the number of tokens in the output places. Without precondition (P2), the Petri-net controller may then reach a state in which a transition is enabled, and the tokens in its input places are consumed without producing tokens in the output places. This would lead to an inconsistent Petri-net because consumption and production of tokens should not be dissociated. Both preconditions are then required in order to be able to call both addToken and removeToken when a transition is enabled. Figure 4.9 gives the B specification of operations addToken and removeToken.

removeToken$(pp) =$
PRE
  $pp \in Place \wedge pp \in \mathbf{dom}(Place\_tokens) \wedge$
  $Place\_tokens(pp) > 0$
THEN
  Place_SetTokens$(pp,\ Place\_tokens(pp) - 1)$
END;

addToken$(pp) =$
PRE
  $pp \in Place \wedge pp \in \mathbf{dom}(Place\_tokens) \wedge$
  $Place\_tokens(pp) < \mathbf{MAXINT}$
THEN
  Place_SetTokens$(pp,\ Place\_tokens(pp) + 1)$
END ;

Figure 4.9: Operations addToken and RemoveToken

Every operation has three preconditions allowing the success of the POs computed by the AtelierB prover:

- the parameter typing predicate $pp \in Place$,
- the definition domain of the tokens number (greater than zero or less than maxint), and
- predicate $pp \in dom(Place\_Tokens)$ which asserts that attribute tokens is assigned to a value for place pp. This predicate guarantees the well-definedness PO of application $Place\_Tokens(pp)$. Indeed, in order to be able to read the value of attribute tokens from a given place, the program must verify that the value is not undefined for that place.

Both operations addToken and removeToken call the basic setter Place_SetTokens (figure 4.6) of the functional model, and hence they preserve the meta-model invariant properties. As the Petri-net running algorithm iterates over input and output places of a transition, we enhance machine semantics by an additional getter (figure 4.10) which returns these sets given a transition tt. Finally, AtelierB discharged four proof obligations from this machine (two POs for the setter call, and two additional POs for the well-definedness of $Place\_Tokens(pp)$) and it was able to prove them automatically.

$$
\begin{array}{l}
src,\ trg \leftarrow getPlaces(tt) = \\
\textbf{PRE} \\
\quad tt \in Transition \\
\textbf{THEN} \\
\quad src := input^{-1}\ [\{tt\}] \\
\quad \|\ trg := output^{-1}\ [\{tt\}] \\
\textbf{END};
\end{array}
$$

Figure 4.10: Operation getPlaces

4.3.4 Semantics coordination

Machine semantics provides B operations that are proved correct. It guarantees that any running algorithm based on these operations will not lead to violations of the model's properties. The notion of algorithm in the oxford dictionary (Oxford, 2020), is defined as: "*an algorithm is a process or set of rules to be followed in calculations or other problem-solving operations*". In order to keep reasoning at a high abstraction level, and be faithful to this definition, operations run and fire presented as algorithms in figure 4.4, will be defined as CSP[6] processes that coordinate operations of machine semantics. The process algebra CSP is an event-based formalism that enables description of patterns of system behaviour. Butler and Leuschel (2005)

[6] CSP: Communicating Sequential Processes Hoare (1985).

propose a combination of CSP and the B method and its integration within ProB. This formalism is then useful for executable DSLs due to its abstraction capabilities and also thanks to its integration within ProB. Figure 4.11 provides the main CSP constructs. In this section we apply the following ones:

- simple action prefix $a \rightarrow P$ where a is a B operation name (called channel in CSP) possibly followed by a sequence of outputs ($!v$) and inputs ($?var$) such that v is a value expression and var is a variable identifier.
- sequential composition ($P \ ; \ Q$) meaning that the execution of process Q follows that of process P.
- process interleaving ($P \ ||| \ Q$) where the resulting execution traces are the arbitrary interleaving of traces issued from each process.
- guarded process ($g : P$) which is the execution of process P under a condition defined by guard g.

Operator	Syntax
stop	$STOP$
skip	$SKIP$
prefix	$a \rightarrow Q$
conditional prefix	$a?x : C \rightarrow P$
external choice	$P \ \Box \ Q$
internal choice	$P \ \sqcap \ Q$
interleaving	$P \vert\vert\vert Q$
parallel composition	$P \ [\![A]\!] \ Q$
sequential composition	$P; Q$

Figure 4.11: Some commonly used CSP operators

Figure 4.12 shows the CSP specification of the petri-net running algorithm. This algortihm is composed of four processes: RUN, FIRE, CONSUME and PRODUCE. Process RUN (line 2.) is a recursion defined by a sequential composition with the prefixed process FIRE. In this sequence channel getEnabled?$trans$ is a call to the B operation getEnabled whose output value is registered in variable $trans$ which is then transmitted to process FIRE. Concretely, variable $trans$ represents an enabled transition provided non-deterministically by operation getEnabled. The simulation of process RUN continues indefinitely or stops when the system reaches a deadlock. Process FIRE, applied to a transition $trans$, is a sequencing of processes CONSUME and PRODUCE preceded by the simple action prefix: getPlaces!$trans$?$input$?$output$. This action is a call to the B operation getPlaces on transition $trans$ in order to get its $input$ and $output$ places which are further transmitted to processes CONSUME and PRODUCE. These two processes apply respectively operations removeToken and addToken to all elements of sets $input$ and $output$. Notation $|||_{[x \in S]}$Op!x represents a replicated interleaving which applies all possible combinations of Op having the various valuations of parameter x taken from set S.

1. MAIN = RUN
2. RUN = getEnabled?$trans$ → FIRE($trans$) ; RUN
3. FIRE($trans$) =
4. getPlaces!$trans$?$input$?$output$ → (
5. CONSUME($input$) ; PRODUCE($output$)
6.)
7. CONSUME($input$) = $|||_{[x \in input]}$removeToken!x → SKIP
8. PRODUCE($output$) = $|||_{[x \in output]}$addToken!x → SKIP

Figure 4.12: CSP formalisation of run and fire

4.4 Debugging with PNet$_{\text{Reference}}$

In Meeduse, ProB and EMF are applied together in order to take benefit of the visualisation capabilities of tools like Sirius and GMF, and the animation and model-checking functions of ProB. In this preliminary study we used the first version of Meeduse (Figure 4.13); it was just a prototype that is used as a proof of concepts.

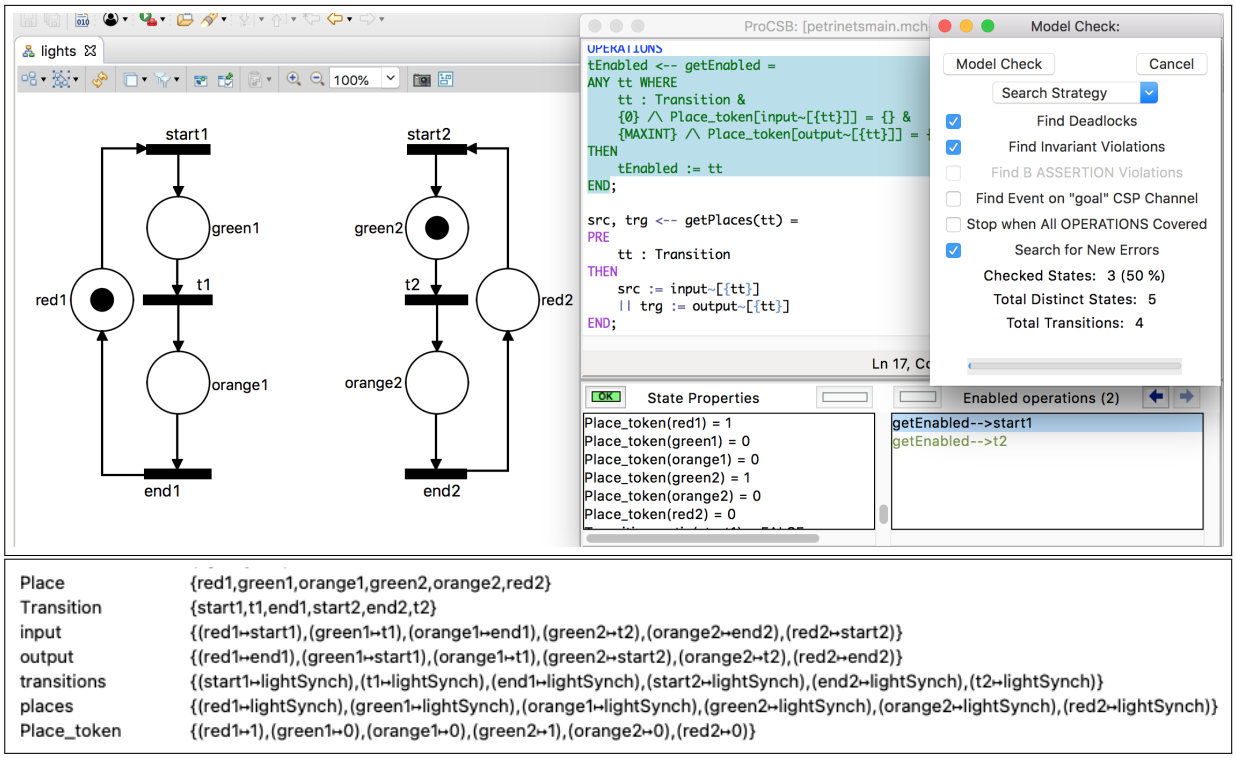

Figure 4.13: Integration of ProB within EMF

EMF and ProB were synchronized at every animation step using a notification mechanism, which allowed us to debug the traffic light via PNet$_{\text{Reference}}$. We have two possibilities to apply the ProB tool:

1. interactive animation, which is useful for domain experts,

2. model-checking, which allows sophisticated analysis of reachability properties.

First, Meeduse interprets an input EMF model, such as the traffic light model of figure 4.2 and injects its elements as valuations in the formal specification of the meta-model. These valuations (bottom part of figure 4.13) allow ProB to initialize the B machine and start animation. The right side of figure 4.13 provides the ProB view and the left side shows our Sirius modeler that we developed for the simplified Petri-net case study. The ProB view shows CSP guided animation of machine semantics. In the current state of the model two operations are enabled: start1 and t2. In interactive animation, depending on the choice done by the user, the tool fires the selected transition and then changes the model according to the formal B specification. For every animation step, Meeduse gets the B machine state from ProB and translates it back to the EMF model in order to update the graphical view. As presented in figure 4.13, ProB offers model-checking functions allowing to find deadlocks, invariant violations and reachability of CSP goals.

4.4.1 Mutual exclusion

The mutual exclusion property can be expressed by the following invariant which excludes a state where traffic lights are simultaneously in their critical sections. A traffic light enters its critical section after enabling transition start and it leaves it by transition end meaning that the critical section includes states Green and Orange:

$$
\begin{array}{l}
1 \in Place\_tokens[\{green1,\ orange1\}] \\
\quad \Rightarrow Place\_tokens[\{green2,\ orange2\}] = \{0\} \\
\wedge\ 1 \in Place\_tokens[\{green2,\ orange2\}] \\
\quad \Rightarrow Place\_tokens[\{green1,\ orange1\}] = \{0\}
\end{array}
$$

In order to check whether this mutual exclusion property holds or not for our Petri-net model, we add this invariant to machine semantics and we ask ProB to find invariant violations starting from several possible initial states. ProB successfully found all the expected invariant violations and also produced the corresponding transition sequences (called counter-examples). ProB provides several interesting facilities to debug a formal specification in order to improve it. Figure 4.14 shows the improved petri-net model where place sync is added between transitions $start$ and end. Note that currently Meeduse does not provide means to apply the feedback produced by ProB (such as trace analysis, invariant decomposition, etc) to the input EMF model. The introduction of place sync is hence done after checking the trace produced by ProB when it found the invariant violation. From the initial state of Figure 4.13 both $start1$ and $start2$ can be enabled because their input places have the required number of tokens. If $start1$ (respectively $start2$) is fired then the token of place sync will be consumed which

forbids Light 2 (respectively Light 1) to enter its critical section. This token will be restored after firing transition $end1$ (respectively $end2$). From this Petri-net model ProB didn't find any invariant violation after visiting all possible nodes of the state space. This proof guarantees the mutual exclusion property of the improved traffic-light.

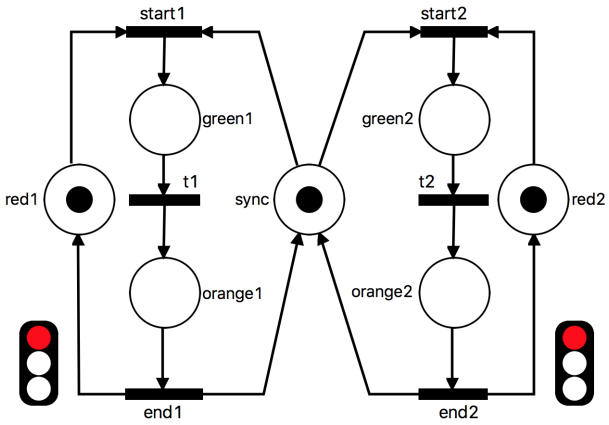

Figure 4.14: Improved petri-net for mutual exclusion (V2)

4.4.2 Fairness

In order to check this property we apply a parallel composition of process RUN with process FAIRNESS defined in figure 4.15, line (12.). This process leads to two possible traces:

- $(step1 \ ; \ step2 \ ; \ goal)$, and
- $(step2 \ ; \ step1 \ ; \ goal)$

Channel $step1$ (respectively $step2$) is produced from process FIRE when guard $trans = end1$ (respectively $trans = end2$) holds. The objective of this specification is to stop the running algorithm when goal STOP is reached, which means that the system produced a trace where both transitions $end1$ and $end2$ are fired by the RUN process. For the example of Figure 4.14 ProB successfully produced the expected sequences leading to the CSP goal process and showing that the system gives fair turns to lights 1 and 2. However, given that the running algorithm is non-deterministic, it would be interesting to seek for the existence of loops where only one light runs. For this purpose, we can override the getEnabled operation in process RUN as follows:

$$\text{RUN} = \text{FIRE}(start1) \ ; \text{FIRE}(t1) \ ; \text{FIRE}(end1) \ ; \text{RUN}$$

From this CSP rule, ProB covered the whole state space and didn't find a deadlock showing that the system may stay running without evolutions of Light 2. This proof exhibits a weak fairness from the model. To improve the traffic light Petri-net the domain expert can introduce

```
1.  MAIN = RUN |[{step1, step2}]| FAIRNESS
2.  RUN = getEnabled?trans → FIRE(trans) ; RUN
3.  FIRE(trans) =
4.        getPlaces!trans?input?output → (
5.              CONSUME(input) ; PRODUCE(output)
6.        ) ;
7.        ( (trans = end1) : step1 → SKIP
8.          [] (trans = end2) : step2 → SKIP
9.          [] (trans ∉ {end1, end2}) : SKIP )
10. CONSUME(input) = |||[x∈input] removeToken!x → SKIP
11. PRODUCE(output) = |||[x∈output] addToken!x → SKIP
12. FAIRNESS = (step1 → SKIP ||| step2 →SKIP) ; goal → STOP
```

Figure 4.15: Fairness checking with CSP

a sequencing mechanism (figure 4.16) and replay the fairness checking, first with the CSP specification of figure 4.15 for non-regression and next with the overriding of rule RUN as mentioned above. The latter ends with a deadlock because after FIRE($end1$) it is not possible to go back and run again FIRE($start1$). The fairness checking with the CSP specification of figure 4.15 provides the same result as previously meaning that this petri-net controller gives fair turns to lights 1 and 2 without any loop where only one light runs.

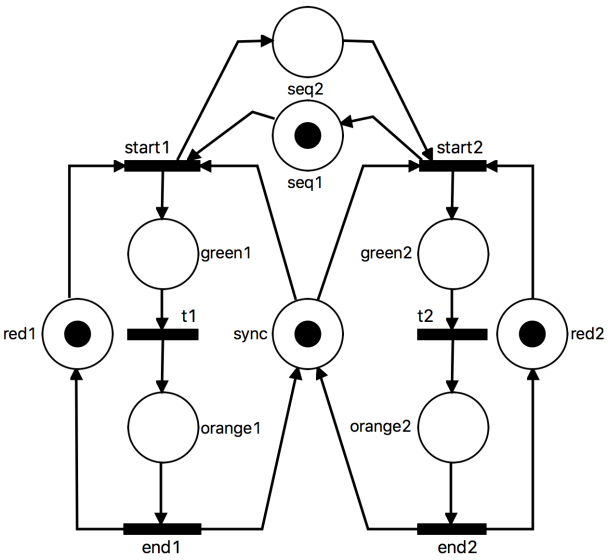

Figure 4.16: Final traffic lights petri-net (V3)

Note that by deleting place sync from figure 4.16 the resulting execution traces, with regard to the elements issued from the initial model, are included in the ones issued from figure 4.14. We can therefore assume that the fairness property is a refinement of the mutual exclusion property, and hence reduce the Petri-net model of figure 4.16 by deleting place sync.

4.5 Conclusion

Few proposals exist for bridging the gap between DSLs with their supporting tools and a proof based formal approach dedicated to execution semantics. The closest ones in comparison with the Meeduse approach in general are (Rivera et al., 2009) and (Gargantini et al., 2010a), where respectively MAUD and Abstract State Machines (ASM) are used. Unfortunately these works do not address the joint execution of the formal model and the DSL, which is a major contribution of Meeduse. One interesting perspective is to integrate several target formalisms within Meeduse being inspired by these works. In this chapter we applied our tool to the simplified Petri-net DSL. In the next chapter we propose several other applications, including PNML, the international standard ISO/IEC 15909 for Petri-nets. Before coming to this realistic application we tried several simplified implementations of this DSL applying MDE tools for DSL execution. The study allowed us to illustrate via several obvious details, how useful is a formal approach especially when addressing a DSL that is dedicated to safety-critical systems.

The major observation is that during the verification activities several concerns must be dissociated: properties of the meta-model, those of the execution semantics and finally those of the coordination algorithm. When an executable DSL is not carefully defined, it may lead to a succession of conceptual failures: failures of modeling operations (*e.g.* setTokens) may result in failures of execution operations (*e.g.* addToken), which in turn may result in failures of the coordination operations (*e.g.* fire/run). The major risk is that when debugging a model, the domain expert does not worry about the correctness of the DSL tool because it is basically the task of the MDE expert. In language programming the developer debugs his/her source-code (in MDE this is seen as a model) using existing IDEs without paying attention to how the language semantics are encoded since in the language theory these semantics are formally defined and compilers are well established techniques. Observations made by this work show that it is difficult to transpose debugging as used in language programming and compilers into executable DSLs especially for safety-critical systems.

Finally, we think that the application of executable DSLs in safety-critical systems require the collaboration of experts from both MDE and formal methods communities. The first actor defines the DSL with associated modeling tools and the second one defines the execution semantics with associated verification activities.

Chapter 5

Applications and case studies

Contents

The systematic mapping study of Iung et al. (2020) shows that among the 59 existing language workbenches (LWBs) only 9 provide supports for verification, which is only done via testing (the formal dimension is completely absent). This is an important shortcoming because the absence of a good support for verification weakens the applicability of DSLs, especially for safety-critical systems. Indeed, in these systems correctness is a strong requirement and it is often addressed by the application of formal methods. During the last three/four years we investigated and developed a solution to this problem by proposing a Formal Model-Driven Engineering (FMDE) approach to create zero-fault DSLs, which are DSLs whose semantics are mathematically proved correct with regards to their structural and logical properties.

We developed the Meeduse tool to make the bridge between MDE and the formal B method. Technically the tool is built on EMF (Steinberg et al., 2009) (The Eclipse Modeling Framework) and ProB (Leuschel and Butler, 2008), an animator and model-checker of the B method that is certified T2 SIL4 (Safety Integrity Level), which is the highest safety level according to the Cenelec EN 50128 standard. ProB is also used by companies such as Alstom, Thales, Siemens, General Electric, ClearSy and Systerel. In addition to the formal reasoning about the semantics of DSLs, Meeduse offers execution and debugging facilities, which is a major contribution in comparison with other works built on DSLs and formal methods. The overall idea is that animation done in ProB is equivalently applied to the input EMF model leading to a correct model execution.

This chapter reviews progress made and three major realistic applications of Meeduse, which attests the interest and power of our FMDE approach to build executable DSLs (called in the remainder xDSLs).

Publications related to the applications of Meeduse

Akram Idani. The B Method Meets Petri-Nets: Shallow and Deep Embedding. *LNCS Transactions on Petri-nets and Other Models of Concurrency*, 2022a. Submitted revised version.

Akram Idani. The B Method meets MDE: Survey, progress and future. In *16th International Conference on Research Challenges in Information Science (RCIS)*, volume 446 of *LNBIP*. Springer, 2022b. URL https://doi.org/10.1007/978-3-031-05760-1_29.

Akram Idani. A Lightweight Development of Outbreak Prevention Strategies Built on Formal Methods and xDSLs. In *ACM European Symposium on Software Engineering (ESSE)*. ACM, 2021. URL https://doi.org/10.1145/3501774.3501787.

Akram Idani. Formal model-driven executable DSLs: Application to Petri-nets. *International Journal on Innovations in Systems and Software Engineering (ISSE)*, 18(4), 2022c. URL https://doi.org/10.1007/s11334-021-00408-4.

Akram Idani, Germán Vega, and Michael Leuschel. Applying Formal Reasoning to Model Transformation: The Meeduse solution. In *Proceedings of the 12th Transformation Tool Contest, co-located with STAF'2019*, volume 2550 of *CEUR Workshop Proceedings*, pages 33–44, 2019a.

Akram Idani, Yves Ledru, Abderrahim Ait-Wakrime, Rahma Ben-Ayed, and Simon Collart-Dutilleul. Incremental Development of a Safety Critical System Combining formal Methods and DSMLs – Application to a Railway System. In *24th International Conference on Formal Methods for Industrial Critical Systems (FMICS'2019)*, volume 11687 of *LNCS*, pages 93–109. Springer, 2019b.

Akram Idani, Yves Ledru, Abderrahim Ait-Wakrime, Rahma Ben-Ayed, and Philippe Bon. Towards a Tool-Based Domain Specific Approach for Railway Systems Modeling and Validation. In *Third International Conference onReliability, Safety, and Security of Railway Systems (RSSRail'2019)*, volume 11495 of *LNCS*, pages 23–40. Springer, 2019c.

Structure

1. Section 5.1: applies Meeduse to the execution semantics of PNML (Petri-Net Markup Language), the international standard ISO/IEC 15909 for Petri-nets, and shows the capabilities of the B method in the Petri-net field. This application led to a full-fledged tool called MeeNET (Meeduse for Petri-Nets).

2. Section 5.2: deals with railway systems (Idani, A. et al., 2019b,a). This application builds on a graphical DSL that can be used by railway experts to design railroad topologies and simulate (un)safe train movements.

3. Section 5.3: addresses model-to-model transformations (Idani, A. et al., 2019c). This application was carried out during the 12^{th} edition of the transformation tool contest (TTC'19) and won two awards: best verification and audience award.

5.1 Application 1: Petri-net Markup Language

The previous chapter showed by practice how a formal model-driven executable Petri-net DSL can be developed, executed and debugged. The discussed example are mainly illustrative and were used to point out a major limitation of MDE tools: the lack of support for verification. Our approach introduces within these tools formal reasoning, which seems highly required especially when the DSL tool, such as a Petri-net based tool, has to be used for safety-critical systems. In the following we present MeeNET[1] our formal Petri-net designer and animator, that is powered by Meeduse and built on top of PNML (Petri-Net Markup Language), the international standard ISO/IEC 15909 for Petri-nets.

5.1.1 PNML

The Petri-net community has worked several years ago on a standardized Petri-net DSL, called PNML (Petri Net Markup Language) (Hillah et al., 2009, PNML Homepage, 2020). The language provides an agreed-on interchange format that is compliant with a formal definition of Petri-nets. The standardization process ISO/IEC 15909 were set up by the community in order to provide a commonly used specification for Petri-nets. The standard covers three classes of Petri-nets: High-Level Petri Nets (HLPN), Symmetric Nets, and Place/Transition Nets (PT-Nets). Models of following sections belong to the third class (PT-nets). Nonetheless our approach can be applied to the two other classes.

We believe that for scalability it is important, for MDE tools and works that are dedicated to executable DSLs, to go beyond the illustrative case study and propose realistic applications based on PNML. Indeed, existing MDE tools have reached a good level of maturity and provide several powerful facilities such as omniscient debugging, trace optimisation, visualization,

[1] Demo videos can be found at: http://vasco.imag.fr/tools/meeduse/meenet/

etc. Furthermore, open-source implementations of PNML and the underlying meta-models are already integrated within EMF. There exist two major platforms: The ePNK ePNK Homepage (2020) and PNML Framework Homepage (2020). In our work, as far as an ECore file of the meta-model is defined, Meeduse can be used, first to extract the formal static semantics from the file and prove its correctness, and then to execute input models if the formal B machine of the operational semantics is provided.

The aim of MeeNET is to provide a scalable executable formal specification of Petri-nets and experiment our approach on realistic input models. The underlying formal specification is currently experimented within The ePNK because this platform provides an extension point, so that new Petri-net types can be plugged in to the existing tool without touching the code of The ePNK. This mechanism is interesting from the Meeduse point of view because it is a kind of DSL refinement, which is a challenging topic for DSL execution and verification. As refinements are apart of the B method, then the tool opens interesting perspectives to this work. We partially address this perspective in the remainder.

5.1.2 The PNML Meta-model

PNML identifies concepts that are common to the three classes of Petri-nets in a so-called PNML Core model (figure 5.1). The common concepts are mainly places, transitions and arcs, and these objects can have some kind of label. The PNML core model provides means for splitting up larger Petri-nets into pages; connections between nodes on different pages can be established by reference places or reference transitions. PNML defines all kinds of graphical information that can be attached to the different elements, such as position, size, font-type, and font-size. Our objective is to formalise the executable part of PNML, so not all the PNML concerns are taken into account. For this reason Meeduse provides an annotation mechanism allowing the DSL developer to select the concepts to be translated into B. This mechanism provides also a way to precise the namings and to strengthen the properties of the meta-model such as multiplicities when required. Note that the current version of MeeNET does not yet cover the splitting of Petri-nets over pages, it is rather focused on the basic constructs. Classes RefPlace and RefTrans, for example, are not transformed into B.

In The ePNK, the PT-Nets class of Petri-nets is defined using an extension mechanism called Petri-net type definition (PNTD). Figure 5.2 shows the corresponding meta-model. It is the purpose of the PNTD to define the specialisations of meta-class Label that are possible in a specific class of Petri-nets, and also to define the additional restrictions on the legal connections. Figure 5.2 introduces two additional kinds of labels for Place/Transition systems: the initial marking for places, and the inscription for arcs. The initial marking can be any natural number (including 0) and the inscription for arcs can be any positive number. Technically, figures 5.1 and 5.2 are two distinct meta-models but one is the extension of the other (PT-Nets is the extension of PNML Core). In our approach, we support this extension mechanism using B refinements.

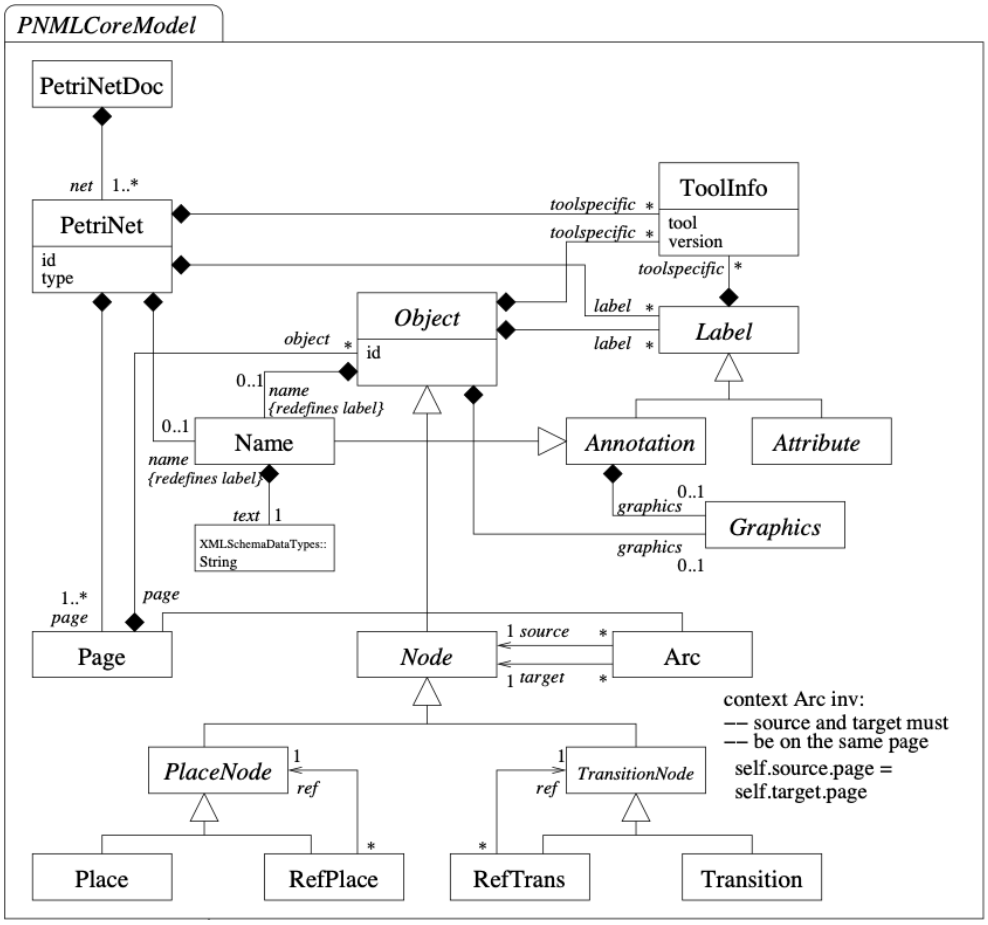

Figure 5.1: The PNML core model of ISO/IEC 15909-2 (2011) (ISO/IEC, 2011)

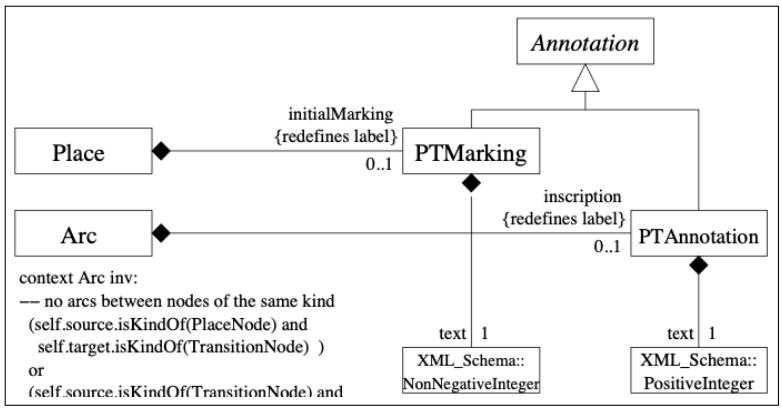

Figure 5.2: The PNTD for PT-Nets (ePNK Homepage, 2020)

5.1.3 Formal static semantics

In B, there are two kinds of refinements: behaviour refinement and data refinement.

- Behaviour refinement means that an algorithm is changed by another one but without vi-

olating neither the conditions under which the initial algorithm is defined nor its possible executions. This kind of refinement would be useful to introduce step by step execution semantics of a DSL. It is not illustrated in this work; we have defined the formal model of the PT-nets execution semantics once and for all.

- Data refinement applies a new set of data in the refined model, that is some data can be replaced and some data can be added. This kind of refinement would be useful to introduce incrementally the static semantics of a DSL. This is suitable for the PT-nets class because its data are defined by introducing some additional meta-classes and by specializing some meta-classes from PNML Core.

Meeduse extracts from every meta-model a specific B machine defining its structural features. In this case study, we get two B machines and then we manually establish a refinement relation between them. Automatic DSL refinements is left to our future works, but we believe that it can be automated by exploiting the annotation mechanism of Meeduse. By this approach, it becomes possible to address the other Petri-net classes without any impact on the Core elements, because every Petri-net class would be defined as a specific refinement of PNML Core.

Figure 5.3 gives the variables and the invariants generated by Meeduse from PNML Core. In this machine the inheritance is translated into set inclusion ($Place \subseteq PlaceNode \subseteq Node \subseteq Object \subseteq ID$). Variable *ID* refers to the objects identifiers. Regarding transitions and places, they are not directly related with input and output references such as in the simplified case study. In PNML Core, they are nodes, and their connections are defined by means of a class Arc.

MACHINE
PNMLCore
SETS
ID_AS;
LABEL
VARIABLES
ID,
Label,
Object,
Node,
Arc,
PlaceNode,
TransitionNode,
Place,
Transition,
source,
target

INVARIANT
$ID \in \mathcal{F}(ID\_AS) \wedge$
$Label \in \mathcal{F}(LABEL) \wedge$
$Object \subseteq ID \wedge$
$Node \subseteq Object \wedge$
$Arc \subseteq Object \wedge$
$PlaceNode \subseteq Node \wedge$
$TransitionNode \subseteq Node \wedge$
$Place \subseteq PlaceNode \wedge$
$Transition \subseteq TransitionNode \wedge$
$source \in Arc \rightarrow Node \wedge$
$target \in Arc \rightarrow Node \wedge$
$Node \cap Arc = \emptyset \wedge$
$PlaceNode \cap TransitionNode = \emptyset$

Figure 5.3: Formal static semantics of PNML Core

The formal static semantics of the PT-nets meta-model is given in Figure 5.4. It is a data refinement of machine PNMLCore introducing several variables. In the meta-model, place markings and arc annotations are defined with the XML data-types NonNegativeInteger and PositiveInteger, as attributes of classes PTMarking and PTAnnotation. The translation into B applies types NAT (natural integers) and NAT_1 (strictly positive natural integers) to the corresponding variables *PTMarking_text* and *PTAnnotation_text*. The initial marking of places and the arc inscriptions are optional (multiplicity 0..1) and cannot be shared (because of the containment). They are then translated into partial injections.

REFINEMENT
ptnets
REFINES
PNMLCore
VARIABLES
PTMarking,
PTAnnotation,
initialMarking,
inscription,
PTMarking_text,
PTAnnotation_text
INVARIANT
$PTMarking \subseteq Label \wedge$
$PTAnnotation \subseteq Label \wedge$
$PTAnnotation \cap PTMarking = \emptyset \wedge$
$initialMarking \in Place \rightarrowtail\!\!\!\!\!\!\!\!\rightarrow PTMarking \wedge$
$inscription \in Arc \rightarrowtail\!\!\!\!\!\!\!\!\rightarrow PTAnnotation \wedge$
$PTMarking\_text \in PTMarking \rightarrow \mathbf{NAT} \wedge$
$PTAnnotation\_text \in PTAnnotation \rightarrow \mathbf{NAT}_1$

Figure 5.4: Formal static semantics of PT-nets

The overall formal specification of the static semantics generated by Meeduse is about 457 lines of code. It provides 38 modeling operations from which the AtelierB prover produced 106 POs (95 were proved automatically and 11 using the interactive prover).

5.1.4 Operational semantics

In order to deal with the operational semantics of the PT-nets class of PNML, we introduce two transient (not serialised) attributes: attribute marking of class Place to represent the current number of tokens in a place and attribute value of class Arc to represent the number of tokens that are consumed and/or produced when transitions are fired. These attributes are single-valued,

mandatory and without an initial value. They are translated into total functions: *Place_marking* and *Arc_value*. Figure 5.5 shows the structural part of the execution semantics machine.

If a place is linked to a PTMarking via relation initialMarking then the corresponding value (represented with variable $PTMarking\_text$) is assigned to its current marking (*initialMarking* ; *PTMarking_text*) in the initialization, otherwise the attribute is set by default at 0 ((*Place* − **dom**(*initialMarking*)) × {0}). The same principle is applied to attribute value of class Arc, but it takes value 1 if the arc does not have an inscription. In fact, when an inscription is not specified on the input arc (respectively the output arc) of a transition, then by default only one token is consumed from its source place (respectively one token is produced into its target place) when the transition is fired. The proof of correctness of this initialization guarantees that none of the place markings and arc values are missed and that they are conformant to their typing domains (NAT and NAT_1).

MACHINE
semantics
INCLUDES
ptnets
VARIABLES
Place_marking, *Arc_value*
INVARIANT
Place_marking ∈ *Place* → **NAT** ∧
Arc_value ∈ *Arc* → $\textbf{NAT}_1$
INITIALISATION
Place_marking := (*initialMarking* ; *PTMarking_text*)
∪ (*Place* - **dom**(*initialMarking*)) × {0}
|| *Arc_value* := (*inscription* ; *PTArcAnnotation_text*)
∪ (*Arc* - **dom**(*inscription*)) × {1}

Figure 5.5: Structural part of the execution semantics

MeeNET integrates Meeduse and the formal static semantics of PT-nets within The ePNK, which allows one to experiment several formal specifications for the execution semantics together with realistic PNML models. For illustration, we propose a different technique than the application of CSP||B. The following formalization (figure 5.6) is inspired by Attiogbe (2009) were Event-B was used to provide a faithful formal semantics to basic and high-level Petri-nets. The notation ⩤ denotes the overriding of a relation by another one. Operation fire selects non-deterministically a transition tt such that all its input places has a sufficient number of tokens; then it overrides relation $Place\_marking$ in order to update the markings of its input and output places. The various definitions used in this operation are given in figure 5.7 and the complete specification is provided in the appendix.

Sets $inputs(tt)$ and $outputs(tt)$ get respectively input and output nodes and arcs of a given transition *tt*. Note that set $inputs(tt)$ is restricted to places from which the consumption is

fire =
  ANY *tt* **WHERE**
    $tt \in Transition \wedge$
    $\mathbf{card}(inputs(tt)) = \mathbf{card}(target^{-1}[\{tt\}])$
  THEN
    $Place\_marking := Place\_marking \lhd\!\!+ consume(tt)$;
    $Place\_marking := Place\_marking \lhd\!\!+ produce(tt)$
  END

Figure 5.6: Operation fire used in MeeNET

possible ($Arc\_value(aa) \leq Place\_marking(pp)$). Consumption and production of tokens are also specified by means of set definitions. Set $consume(tt)$ (respectively $produce(tt)$) computes the new marking of the input (respectively output) nodes of a transition tt.

$$inputs(tt) == \{pp, aa \mid pp \in Place \wedge aa \in Arc \wedge source(aa) = pp \wedge target(aa) = tt \wedge Arc\_value(aa) \leq Place\_marking(pp)\}$$

$$outputs(tt) == \{pp, aa \mid pp \in Place \wedge aa \in Arc \wedge target(aa) = pp \wedge source(aa) = tt \wedge Place\_marking(pp) + Arc\_value(aa) \leq \text{MAXINT}\}$$

$$consume(tt) == \{pp, nn \mid pp \in \mathbf{dom}(inputs(tt)) \wedge nn = Place\_marking(pp) - Arc\_value(inputs(tt)(pp))\}$$

$$produce(tt) == \{pp, nn \mid pp \in \mathbf{dom}(outputs(tt)) \wedge nn = Place\_marking(pp) + Arc\_value(outputs(tt)(pp))\}$$

Figure 5.7: Definitions

5.1.5 MeeNET

Figure 5.8 is a screenshot of MeeNET while executing a PNML file of a satellite memory system. This file is taken from the benchmark of PNML files provided by the 10th edition of the model-checking contest (MCC'2020)[2]. We designed the graphical and the tabular views in Sirius in order to visualize graphically the model. For example, the green transitions (t1 and t4) are transitions that can be fired in the current execution step. They satisfy the guard of operation fire. The animation view and the state view represent the output of ProB. The animation view is for interactive debugging, it allows the user to select which transition to fire; and State view gives the current valuations of the B variables of the static semantics machine.

[2] The benchmark can be found at: https://mcc.lip6.fr/models.php

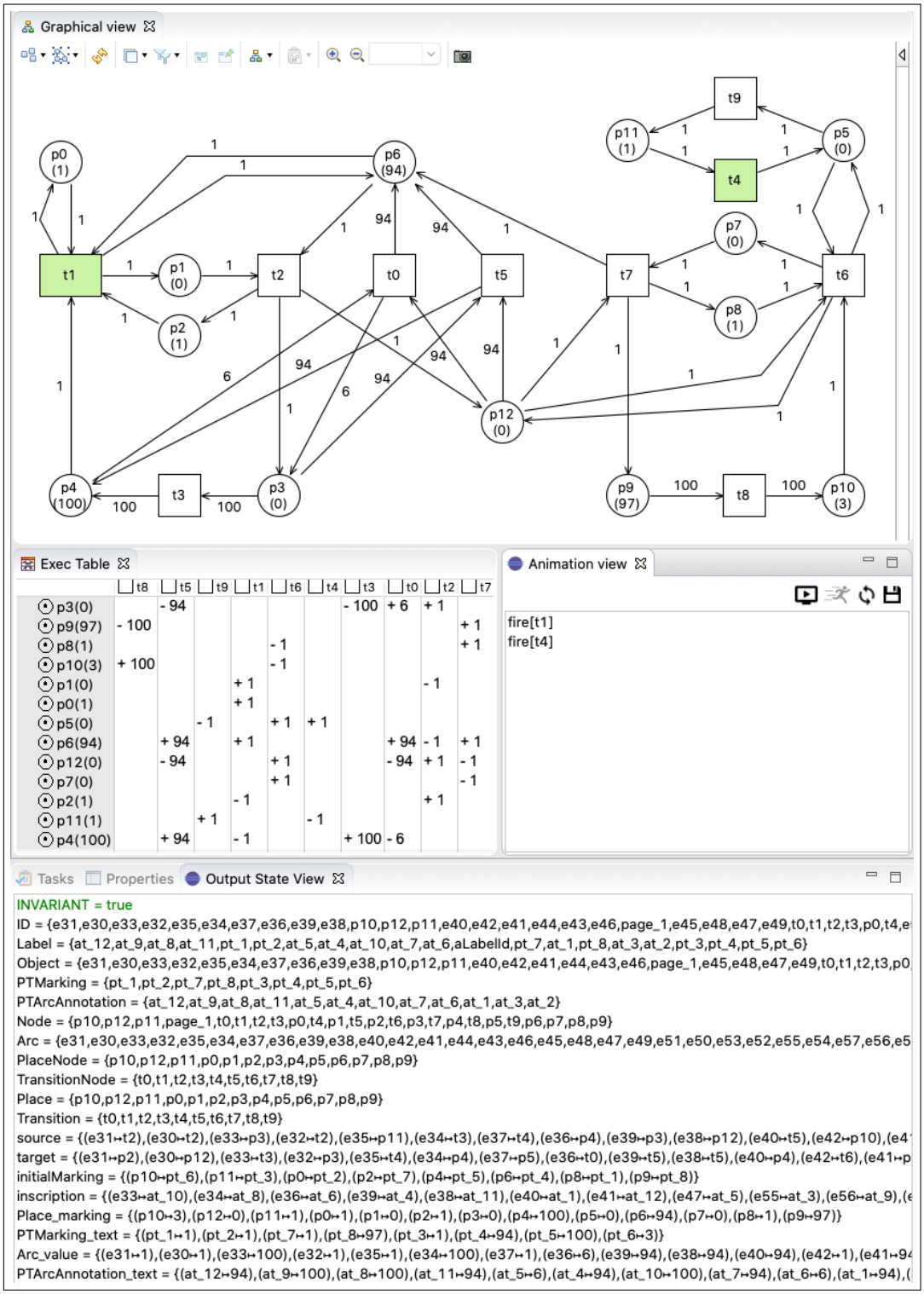

Figure 5.8: A satellite memory PNML model

The Petri-net of Figure 5.8 models the behaviour of the mass memory management system in a micro-satellite from the Myriade product line, designed by CNES, the French Space

Agency. The corresponding controller must ensure that there is always a "security buffer" between the sectors pointed by places p9 and p3. The size of this buffer is a parameter Y (whose value in this model is equal to 3). The system should guarantee the following invariant:

$$p3 > p9 \Rightarrow p3 - p9 \geq Y$$

Based on the valued B specification produced by MeeNET, the ProB model-checker explored all the state-space and showed that this property is preserved all along the execution of the Petri-net model. Other properties, such as LTL properties, are provided with the model and ProB was also able to check them.

We have tried several PNML files from the MCC'2020 in MeeNET (even the biggest ones) and it was able to produce a valued B specification that is further used in ProB for verification purposes. Nevertheless, our objective is not to evaluate the verification capabilities of ProB or its performance based on the resulting B models. The contribution of our work is that it made possible the use of ProB in the Petri-nets field. Further works and experiments are required if one would like to apply ProB to compete with established Petri-net based model-checkers.

5.2 Application 2: Railway systems

The emergence of DSL tools in safety-critical systems (James et al., 2013, Svendsen et al., 2012, Vu et al., 2014) allows domain experts themselves to provide useful structural models to the software system engineer who will then develop the operational aspects of the system. However, as far as we know, none of the existing works in the safety-critical domain, proposes a way to define proved formal links between the behavioural aspects of a system (in Petri-nets or other well known formalisms) and DSL tool development. In this section, we give an overview the work that we presented in (Idani, A. et al., 2019b). We start from an intuitive description of a safety critical system where the operational aspects are specified thanks to high-level Petri-nets, especially coloured Petri-nets (CP-nets), and the structural aspects are designed in a dedicated DSL. The challenge is therefore to merge both worlds (that of CP-Nets and that of DSLs) and then apply AtelierB in order to prove the correctness of the resulting system. Figure 5.9 gives the overall architecture that we propose to adress this challenge.

The DSL meta-model and CP-Net models are automatically translated into B specifications which are enhanced by safety invariants and proved. Then, our approach defines linkage machines allowing to control the functional model and the associated DSL-tool thanks to the CP-Net specifications. Every linkage machine refines a CP-Net model and includes the functional model. This technique is different from the one discussed in section 5.1 for basic Petri-nets, since the objective is not to execute CP-nets, but rather use CP-nets in order to define the operational semantics of a given DSL. Indeed, by executing a CP-Net model, Meeduse plays with the DSL and verifies, via ProB, that the linkage invariants are preserved all along the animation.

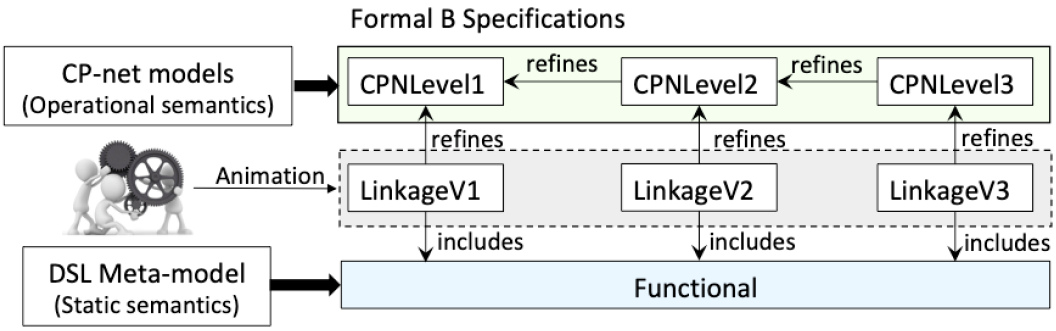

Figure 5.9: Merging CP-Nets and DSLs in a formally defined framework

5.2.1 ERTMS/ETCS Case Study

This work was funded by the NExTRegio project of IRT Railenium. The project aims at performing a system level analysis of a railway signalling system taking into account emergent solutions for train automation. Indeed, in the last decade, new technologies have been considered in railway systems in order to improve automation on the one hand and to reduce the operating costs on the other hand. In particular, the European ERTMS/ETCS[3] (ERA, 2016, Schon et al., 2014) has emerged to replace various national signalling systems. There are three levels of ERTMS/ETCS which differ by the used equipments and the operating mode. The first two levels are already operational. However, ERTMS Level 3 is still in design and experimentation phases: it aims at replacing signalling systems by a global european one which is a GPS-based solution for the acquisition of train positions. In 2018, the ABZ conference (Butler et al., 2018), which gathers several formal methods communities, proposed a case study[4] to model ERTMS/ETCS level 3 and has published several formal models. Unfortunately, these models do not combine the power of formal methods with domain specific approaches and hence they favour verification rather than domain expert validation. The application presented in this section contributes to the design phase of ERTMS/ETCS level 3 by mixing formal techniques and domain specific modeling in a well-known Model Driven Engineering (MDE) paradigm which makes easier domain expert validation.

An ERTMS Level 3 solution is based on train position and train integrity confirmation, both transmitted by the on-board train system (called EVC[5]) to the trackside system (called RBC[6]). Given this information, the traffic agent, via RBC, assigns a movement authority to a train allowing it to move to a given point. In the RBC, track-circuits exist in a logical form by means of trackside train detection sections (called TTD) which are in turn divided into virtual subsections (called VSS). Figure 5.10, taken from the ERTMS 3 reference document (ERA, 2016), illustrates a track circuit divided into two TTDs and four VSSs, and where a train is located on

[3] ERTMS: European Rail Traffic Management System.
ETCS: European Train Control System.
[4] https://www.southampton.ac.uk/abz2018/information/case-study.page
[5] European Vital Computer.
[6] Radio Block Center.

VSS23. This simplified view of section conventions, used by railway experts, applies specific domain representations to represent a situation where a train went through TTD2 and reached its ending VSS.

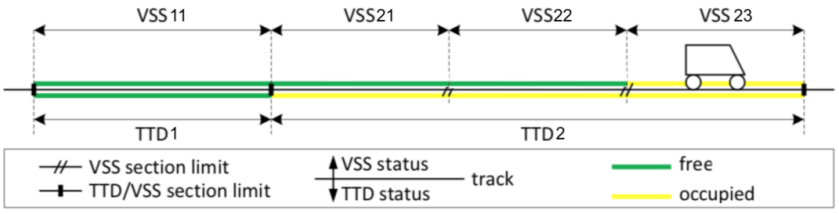

Figure 5.10: Section conventions ERA (2016)

5.2.2 Coloured Petri-nets

Several high-level variants of Petri-nets, like coloured petri-nets or predicate-transition nets, were applied in safety-critical systems and were assisted by formal verification techniques such as animation, model-checking or proofs. Moreover, some experiences like that of the Oslo subway, reported in (Hagalisletto et al., 2007), show that in addition to their formal semantics, high-level Petri-nets favoured communication with domain experts, because chief engineers from railroad infrastructure and traffic department who are not specialists in Petri-nets nor in formal methods, were not only able to understand the models, but also to suggest improvements.

We use coloured petri-nets (CP-Nets), which abstract away structural constraints and focus on safety-critical behaviours. CP-Nets combine the strengths of classical Petri-nets with the strengths of high-level programming languages (Gehlot and Nigro, 2010), to allow handling data types with pre-defined functions. For a formal description of CP-nets, one can refer to (Jensen, 1981); nonetheless, the main concepts used here are:

- Data types: can be simple types (*i.e.* Integer, Boolean,...) or complex types (*i.e.* arrays, sequences,...). In this work, we mainly use integer enumerations.
- Places: represent abstractions on data values (called tokens or colours). The place type is called the colour set and it is defined by composing data-types.
- Transitions: they are linked to input and output places. When fired, a transition consumes tokens from its input places such that they match the transition signature. Then, the transition introduces tokens into its output places.
- Predefined functions: describe some computations done by the transitions when they are fired. In this paper, we use three basic functions: calculation of the next (n^{++}) and the previous value (n^{--}) given a token n when n is of type integer, and the negation value ($\neg n$) when n is of type boolean.

5.2.2.1 Level 1: Simple train movements

Our first CP-net (figure 5.11) defines simple train movements without train integrity nor movement authorities. This abstract level is mainly intended to guarantee the absence of accidents.

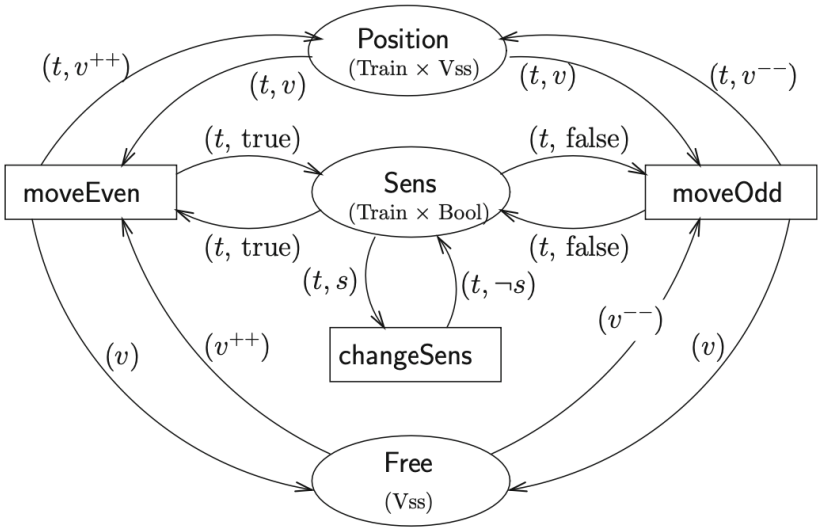

Figure 5.11: Simple movement described in a CP-net

This model describes train movements using transitions *moveEven* and *moveOdd* which move the train forward or backwardand; and *changeSens* which switches the train moving direction. Place *Position* contains pairs (t, v) which record the current VSS v occupied by a train t. Place *Free* gathers the sections which are not occupied by any train and place *Sens* registers for every train its current direction. For our first CP-net model, we would like to prove five safety properties:

1. Absence of accidents meaning that at most one train occupies a Vss,
2. Every train is located in one and only one Vss,
3. Absence of overlapping between Vss states free and occupied,
4. Vss states cannot be undefined, they are either free or occupied,
5. The train moving direction is never lost

Transition *moveEven* is fired given a train t located on section v, whose direction is set to $true$, and such that its next section v^{++} is free (*e.g.* $(t, v) \in Position \wedge (v^{++}) \in Free$). When fired, this transition instantly moves train t from section v to section v^{++}. It consumes tokens (t, v) and (v^{++}) respectively from places *Position* and *Free*, and then respectively introduces into these places tokens (t, v^{++}) and (v), meaning that v^{++} becomes the new position of train t, and section v is released. Transition *moveOdd* applies the same principles to trains in direction $false$ but selects the previous section v^{--} if this section is free. Transition *moveEven* is

fired provided that the train direction is set to $true$ while transition *moveOdd* is fired when its direction is $false$.

5.2.2.2 Extraction of B specifications

In order to prove the safety properties of our first level CP-net model we translate it into B specifications as follows:

MACHINE $CPNData$ **CONSTANTS** $maxTrain$, $maxVss$, $CPNTrain$, $CPNVss$ **PROPERTIES** $maxTrain \in \mathbf{NAT} \wedge maxVss \in \mathbf{NAT}$ $\wedge\ CPNTrain = 1 \,..\, maxTrain$ $\wedge\ CPNVss = 1 \,..\, maxVss$	**REFINEMENT** $CPNLevel1$ **REFINES** $CPNData$ **VARIABLES** $Free$, $Position$, $Sens$ **INVARIANT** $Free \subseteq CPNVss$ $\wedge\ Position \subseteq CPNTrain \times CPNVss$ $\wedge\ Sens \subseteq CPNTrain \times \mathbf{BOOL}$

First an abstract machine (named *CPNData*) is generated in order to gather the colour sets together with the transition signatures as defined in the CP-net model. Colour sets *Train* and *Vss*, which are integer enumerations, are translated into bounded natural constants *CPNTrain* and *CPNVss*. Places *Free*, *Position* and *Sens* become variables in refinement *CPNLevel1* because their values evolve during the execution of the CP-net. In this refinement, by default the variable typing applies general functions such as sets' cartesian product and inclusion (*e.g.* $Position \subseteq CPNTrain \times CPNVss$).

Every transition leads to a basic operation defined in machine *CPNData* with a typing precondition and a *skip* substitution, like the example below of operation *moveEven*:

```
/* Operation moveEven in machine CPNData */
moveEven(tt, vv) =
   PRE tt ∈ CPNTrain ∧ vv ∈ CPNVss THEN
      skip
   END
```

The *skip* substitution of the basic operations is then refined in *CPNLevel1* by introducing the enabledness guards and the expected actions of the transition. In the following we give the refinement of operation *moveEven* in *CPNLevel1*:

```
/* Refinement of the skip substitution in CPNLevel1 */
moveEven(tt, vv) =
   SELECT
      (tt ↦ vv) ∈ Position ∧ (vv + 1) ∈ Free ∧ (tt ↦ TRUE) ∈ Sens
   THEN
      Free := (Free − {(vv + 1)}) ∪ {(vv)} ||
      Position := (Position − {(tt ↦ vv)}) ∪ {(tt ↦ vv + 1)}
   END ;
```

Transitions *moveOdd* and *changeSens* are translated by applying the same principles. Regarding the five safety properties, they are manually introduced in machine *CPNLevel1* using the following invariants:

$$
\begin{array}{l}
Position \in CPNTrain \rightarrowtail CPNVss \ \texttt{/* Properties (1) and (2) */} \\
Free \cap \mathbf{ran}(Position) = \emptyset \ \texttt{/* Property (3) */} \\
Free \cup \mathbf{ran}(Position) = CPNVss \ \texttt{/* Property (4) */} \\
Sens \in CPNTrain \rightarrow \mathbf{BOOL} \ \texttt{/* Property (5) */}
\end{array}
$$

These invariants restrict the state space defined by the typing predicates presented above. For example, the typing predicate of relation *Position* defines all combinations of CPNTrain and CPNVss couples, while the invariant restricts these combinations to those where a CPNTrain is linked to one and only one CPNVss while a CPNVss is linked to at the most one CPNTrain. In our methodology, we consider that if the CP-net model is correct, proofs should be done without any enhancement of the corresponding B specifications. Otherwise, we decide whether the CP-net model is wrong or not, given the AtelierB feedbacks. In all cases we do not modify the generated B operations; we either call the interactive prover when the proof fails due to a limitation in the automatic prover, or we correct the CP-net model and translate it again into B. The initial marking substitutions are introduced without invariant violation:

$$
\begin{array}{l}
\mathbf{INITIALISATION} \\
\quad Position :\in CPNTrain \rightarrowtail CPNVss \ ; \\
\quad Free := CPNVss - \mathbf{ran}(Position) \ ; \\
\quad Sens :\in CPNTrain \rightarrow \mathbf{BOOL}
\end{array}
$$

Based on machines *CPNData* and *CPNLevel1*, and these additional invariants, the AtelierB generated 17 proof obligations and automatically proved 11 amongst them. The 6 other POs were proved using the interactive prover.

5.2.3 A Railway DSL for ERTMS/ETCS

5.2.3.1 The meta-model

In order to provide a tool for domain experts allowing them to draw models like that of figure 5.10, we apply model-driven engineering tools for DSML creation (EMF, Ecore-Tools and Sirius). In MDE, the creation of a DSML starts by the definition of its meta-model and then for every class in the meta-model a graphical representation is created. Figure 5.12 gives the meta-model that we use in this work and figure 5.13 gives a screenshot of the resulting DSML-tool in which a model is designed using the proposed graphical representations.

In our meta-model, a railway system is composed of trains (class *Train*), track sections called TTD in ERTMS/ETCS 3 (class *Trackside*) and which are divided into portions called VSS (class *VirtualBlock*). The bottom of figure 5.13 draws an overall railway topology by means of TTD links. Every portion of a given TTD may be linked to two next and previous portions

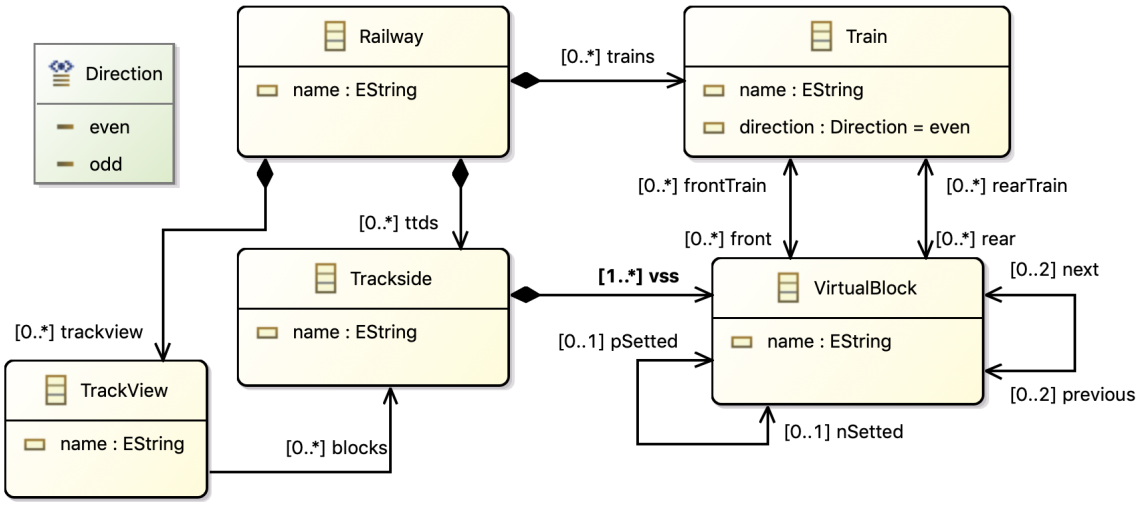

Figure 5.12: A railway meta-model

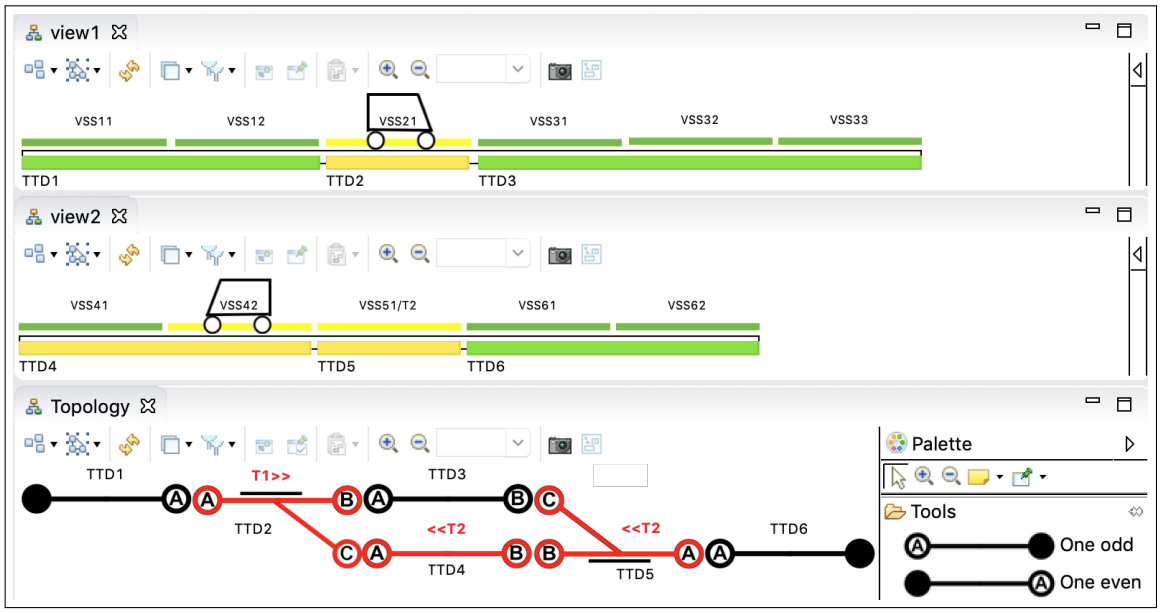

Figure 5.13: A railway model

at the most. In practice, there are four kinds of portions: track extremity (*e.g. VSS11* and *VSS62*), middle track (*e.g. VSS12*), switch (*e.g. VSS21* and *VSS51*) and diamonds. Association *pSetted/nSetted* provides the currently selected previous/next portion among those to which a portion is linked. This is useful especially for switches and diamonds. For example, the next portions of *VSS21* are *VSS31* and *VSS41*, but the position of the switch sets the currently selected next portion of *VSS21* to *VSS31* and hence the selected previous portion of *VSS31* is *VSS21* but for portion *VSS41* there is no previous selected portion. Portion *VSS41* remains then a track limit until the switch position is changed. Note that relation *pSetted/nSetted* is independent from train direction and a track limit is a portion without a selected next or previous portion.

Class *TrackView* represents linear views that follow the current next/previous selections and where every view starts and ends with track limits. For example, the topology presented in the bottom of figure 5.13, leads to the two views on the top of the figure. The first view covers sections *TTD1*, *TTD2* and *TTD3* and the second view covers the three other sections: *TTD4*, *TTD5* and *TTD6*. If the switches position changes, these views are changed consequently. For example, if the selected next portion of *VSS21* is set to *VSS41*, then the resulting topology would lead to two different views: one composed of *TTD1*/*TTD2*/*TTD4*/*TTD5*/*TTD6*, and an other view dedicated to *TTD3* only.

Trains have a direction (*even* or *odd*) and their representation depends on the set of portions that their head and rear occupy. In the example of figure 5.13 we consider two trains: T1 whose front and rear occupy the same portion (*i.e. VSS21*), and T2 that stretches from portion *VSS42* to *VSS51*. A TTD is occupied when at least one of its portions are occupied. This is represented by the yellow color in the track views and by the red color in the topology representation. The green color is used to represent free TTD and VSS in the track view.

5.2.3.2 Formal model

We apply Meeduse to provide domain experts with a formally defined DSL-tool. We give below the translation of classes *Train* and *VirtualBlock* and one basic operation *Train_AddFront* which adds a virtual block to the set of virtual blocks occupied by the head of a train. Several other basic operations are generated by the tool like: *Train_RemoveFront*, *Train_AddRear*, *Train_RemoveRear*, etc.

MACHINE *Functional*
SETS
VIRTUALBLOCK; *TRACKSIDE*
Direction = {*even*,*odd*};
VARIABLES
Train, *VirtualBlock*, *Train_direction*,
frontOfTrain, *rearOfTrain*
INVARIANT
$Train \subseteq TRAIN$
$\wedge\ VirtualBlock \in VIRTUALBLOCK$
$\wedge\ frontOfTrain \in Train \leftrightarrow VirtualBlock$
$\wedge\ rearOfTrain \in Train \leftrightarrow VirtualBlock$
$\wedge\ Train\_direction \in Train \rightarrow Direction$

Train_AddFront(*aTrain*,*aFront*) =
PRE
$aTrain \in Train \wedge$
$aFront \in VirtualBlock \wedge$
$(aTrain \mapsto aFront) \notin frontOfTrain$
THEN
$frontOfTrain :=$
$\quad frontOfTrain \cup \{(aTrain \mapsto aFront)\}$
END;

Machine Functional generated by Meeduse is about 500 lines with 38 basic operations from which the AtelierB produced 80 proof obligations that were proved automatically. Given a model (like that of figure 5.13) Meeduse injects it as valuations in the B specification and calls ProB in order to compute the list of operations that may be animated from these valuations. For

example, the following initialization is extracted by Meeduse from our graphical model. The resulting initial state is hence equivalent to the domain model.

INITIALISATION
$Train := \{T1, T2\} \;||$
$VirtualBlock := \{VSS11, VSS12, \ldots, VSS62\} \;||$
$frontOfTrain := \{(T1 \mapsto VSS21), (T2 \mapsto VSS42)\} \;||$
$rearOfTrain := \{(T1 \mapsto VSS21), (T2 \mapsto VSS51)\} \;||$
$Train\_direction := \{(T1 \mapsto even), (T2 \mapsto odd)\}$

Starting from this initial state, when the user asks Meeduse to animate a B operation, the tool calls ProB and gets the new variable valuations and then it translates back these valuations to the graphical model. This technique results in an automatic visual animation of domain models. For example, the animation of operation *Train_AddFront(T1, VSS31)* introduces couple $(T1 \mapsto VSS31)$ into relation *frontOfTrain* and then Meeduse modifies the domain model as presented in figure 5.14 where the head of T1 occupies two virtual blocks *VSS21* and *VSS31*. Since *VSS31* is one of the portions of *TTD3*, then the visual representation of *TTD3* automatically changes from green to yellow.

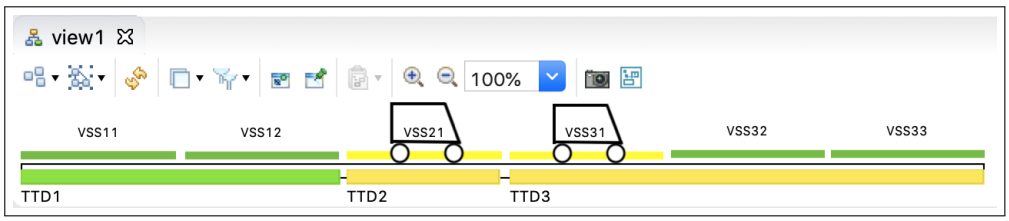

Figure 5.14: View 1 after animation of *Train_AddFront(T1, VSS31)*

5.2.4 Putting it all together

Section 5.2.2 focused on train behaviours with an abstract Petri-net specification that guarantees the absence of accidents, and section 5.2.3 focused on domain modeling of structural aspects of a railway DSL. In this section, we combine both concerns in order to provide a railway DSL with a proved safe train behaviour. The B specifications extracted from the meta-model of Figure 5.12 represent formal static semantics of our DSL, and those extracted from a CP-net model introduce its operational semantics. In order to merge static and operational semantics we create machine *LinkageV1*; it refines *CPNLevel1* and includes machine *Functional*:

REFINEMENT *LinkageV1*
REFINES *CPNLevel1*
INCLUDES *Functional*
VARIABLES
trainMapping, *vssMapping*, *view*
INVARIANT
$trainMapping \in Train \rightarrowtail\!\!\!\!\!\!\rightarrow CPNTrain$
$\wedge\ vssMapping \in VirtualBlock \rightarrowtail\!\!\!\!\!\!\rightarrow CPNVss$
$\wedge\ view \in TrackView$

The refinement guarantees the preservation of the safety invariants of *CPNLevel1* and the inclusion allows us to redefine the CP-net transitions and data using the functional variables of the DSL. In this machine the linkage of the DSL and the CP-net model is done via functions *trainMapping* and *vssMapping*. They respectively map variables *Train* and *VirtualBlock* issued from the meta-model to sets *CPNTrain* and *CPNVss* issued from the CP-net. In our approach every view in the DSL is controlled by a CP-net since the CP-net defines the VSS set by a sequence of integers. Then, the mapping functions are applied to a given *view* (*view* $\in$ *TrackView*). For example, the *vssMapping* relation is computed in the initialisation of *LinkageV1* as:

LET *mapVss* **BE** $mapVss = \mathbf{ran}((\{view\} \lhd blocks^{-1}\ ;\ theVSSs^{-1}))$ **IN**
ANY *map* **WHERE**
$map \in mapVss \rightarrowtail CPNVss \wedge$
$\forall\ vss\ .\ (vss \in mapVss \wedge nSetted[\{vss\}] \neq \emptyset$
$\Rightarrow nSetted(vss) \in \mathbf{dom}(map) \wedge map(nSetted(vss)) = map(vss) + 1)$
THEN
$vssMapping := map$
END
END

Note that *blocks* and *theVSSs* represent respectively association blocks between classes TrackView and Trackside, and association vss between classes Trackside and VirtualBlock. Local variable *mapVss* defined by: $\mathbf{ran}((\{view\} \lhd blocks^{-1}\ ;\ theVSSs^{-1}))$ extracts the set of VSS for a given *view* and the mapping is a total injection ($\rightarrowtail$) that maps every VSS in this view to a unique value from set *CPNVss*. This mapping is done under the condition that if a VSS is not a track extremity ($nSetted[\{vss\}] \neq \emptyset$) then its next selected VSS is mapped ($nSetted(vss) \in \mathbf{dom}(map)$) and the associated CP-net value is equal to the VSS value plus one. We similarly compute the *trainMapping* relation but under the condition that only trains whose head and rear occupy the same VSS are mapped. In this sense, from the example of figure 5.13 only the first view can be mapped and then controlled by our first level CP-net model.

Given the mapping relations, the safety invariants of *CPNLevel1* are rewritten by means of linkage invariants ensuring the relationship between the various B specifications. For example, invariant $Free \cap \mathbf{ran}(Position) = \emptyset$ used for `Property (3)` becomes as follows:

$$(frontOfTrain \cup rearOfTrain)^{-1}[vssMapping^{-1}[Free]] = \emptyset$$

This means that for every free VSS in the CP-net model, the corresponding virtual block in the DSML does not contain any train head or rear. Having the linkage invariants, operation *moveEven(tt, vv)* in the linkage machine is applied to a train mapped to tt, whose head and rear occupy a VSS mapped to vv, and whose direction is $even$ and such that the next VSS which is mapped to $vv + 1$ is free. Actions of *moveEven* call basic functional operations issued from machine *Functional*. They simply remove the head and the rear of the train from vv and put them on $vv + 1$. In the following we give the refinement of operation *moveEven* in *LinkageV1*:

moveEven(*tt*, *vv*) =
LET *train*, *vss*, *nextVss* **BE**
  train = $trainMapping^{-1}(tt)$
  $\wedge$ *vss* = $vssMapping^{-1}(vv)$
  $\wedge$ *nextVss* = $vssMapping^{-1}(vv + 1)$
IN
  SELECT
    $(train \mapsto vss) \in frontOfTrain \cap rearOfTrain$
    $\wedge$ $nextVss \notin$ **ran**$(frontOfTrain \cup rearOfTrain)$
    $\wedge$ $Train\_direction(train) = even$
  THEN
    Train_RemoveFront(*train*, *vss*); **Train_AddFront**(*train*, *nextVss*) ;
    Train_RemoveRear(*train*, *vss*); **Train_AddRear**(*train*, *nextVss*)
  END
END ;

41 POs were generated and proved by the AtelierB for machine *LinkageV1*, which mean that the safety properties (those of machine *CPNLevel1*) as well as the structural properties (those of machine *Functional*) are preserved. Regarding the execution of the DSL, it is done by railway experts using the animation facility of Meeduse. By animating operations from the linkage machine, Meeduse automatically updates the corresponding graphical model leading to its execution.

5.2.5 Refinements

We also apply the refinement principle of the B method to incrementally define formal operational semantics by means of refined CP-net models. In this section, we only focus on the first refinement level; we refer the reader to (Idani, A. et al., 2019a,b) for further details. The general idea is that at every refinement step we introduce additional conceptual elements with associated safety properties and we prove the preservation of these properties as well as those of the previous level:

- Level 2 (authorized train movements): The assumption made in the first CP-net level, considering that a train moves to the next free virtual section and immediately leaves its current section, is quite simplistic but sufficient in order to model an abstract accident-free behaviour. In this second level we introduce a movement authority mechanism, in order to construct routes to which trains are allowed to move. The additional safety invariants of this second CP-net level are: (6.) a VSS cannot be waiting and at the same time assigned to a movement authority; and (7.) a movement authority cannot be shared by several trains. Given CP-net of Level 2 and the corresponding safety properties, as well as the refinement invariant, the AtelierB prover generated 32 POs, such that 25 were proved automatically and 7 interactively, which means that CP-net Level 2 guarantees its own properties and also those of CP-net Level 1.

- Level 3 (movements with integrity confirmation): In the third refinement level we consider a more realistic train representation than that developed in the two previous levels where a train occupies only one VSS. In this refinement, a train is seen as a logical entity defined by the set of VSS that it occupies: its head (place *Position*), a set of VSS not yet released behind its head (place *Wagon*) and the safe rear end (place *Tail*) which is in our case one additional VSS defining the minimal distance between two trains. Thus, a train occupies at least two virtual sections: one for its head and one behind it. When a train moves, its head is advanced from its current VSS v to the next VSS v^{++}, and then v is not freed but a virtual wagon is created over it. Indeed, in ERTMS/ETCS 3, the train must confirm its integrity (*i.e.* it did not lose wagons) before releasing its safe rear end which advances its tail by one VSS and removes the corresponding virtual wagon. Given the B specifications issued from this third level and the associated safety invariants, the AtelierB produced 62 POs and automatically proved 41 among them. The 21 other POs were proved manually.

5.3 Application 3: Model transformation

This section summarizes the application of Meeduse to the 2019 edition of the Transformation Tool Contest (Garcia-Dominguez and Hinkel, 2019) and gives the lessons learned from this study[7]. Among other challenges, the contest emphasizes on correctness which motivated us to apply Meeduse. This study allowed us to try how far we can push the abilities of a formal method to be integrated within model-driven engineering. The results showed that Meeduse can be adapted to model transformation which brings to this field formal automated reasoning tools like AtelierB for theorem proving and ProB for model-checking. Meeduse, combined with ProB, provides three strategies: random animation, interactive animation and model-checking. The first strategy runs randomly the transformation rules until it consumes all the elements

[7] The proposed solution and demonstration videos can be found at: https://github.com/meeduse/Meeduse_TTC_2019.

of the input model and produces the output model. The second strategy applies a step-by-step debugging of the transformation rules, and the third strategy is useful for analysing the reachability of some defined states which allows one to verify whether unwanted situations may happen or not.

Our main objective for the TTC'2019 challenge is to reuse some selected components of Meeduse in order to execute a formal specification of transformation rules given realistic input models. As Meeduse was not initially designed to define model transformations, but to define executable DSLs, we need to rethink the model transformation problem in terms of operational semantics of an abstract machine. However, presenting the full B model of the TTC'2019 case study requires introducing many concepts and properties, which would be inconvenient for readers and take much space in this document. Hence, we decided to introduce in this section our approach and its underlying formal concepts step by step using a simplistic model transformation example inspired by some basic MDE material available on-line. Application of Meeduse to the TTC'2019 case study is described in (Idani, A. et al., 2019c).

Figure 5.15 shows simple input and output meta-models for the example transformation. The input is a model defining persons (of any gender) which may be married (represented by the person1/person2 relation) and the output is a model focused only on married wives and husbands.

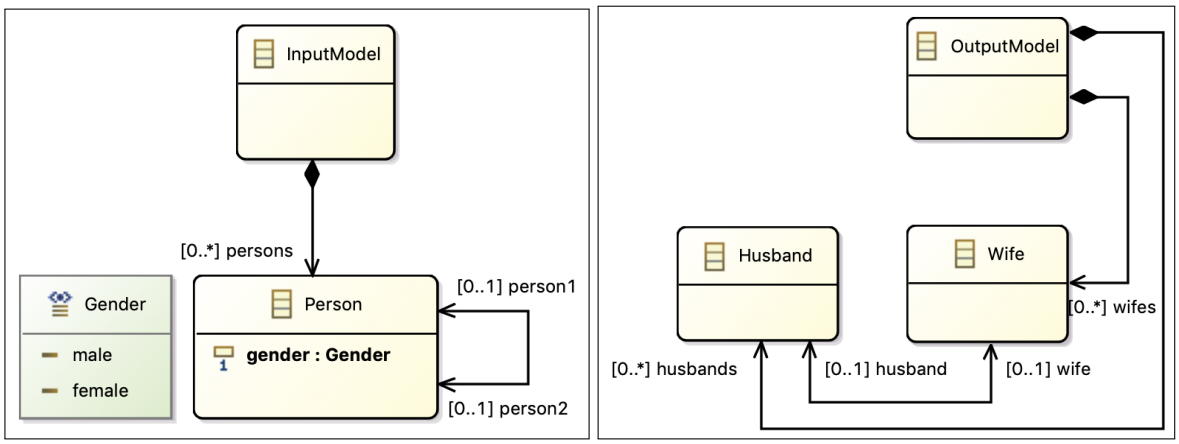

Figure 5.15: Input/Output meta-models

5.3.1 Step 1: merging meta-models

The input of our tool is the meta-model of a DSL. In order to apply it for model-to-model transformation, our idea is first to merge both input and output meta-models into a single one that is automatically translated by Meeduse into B, and then specify the transformation rules as operations of the corresponding B machine. Intuitively, the state of the transformation execution includes the input model, the partially generated output model and any additional information required by transformation rules (for instance, traceability links created by other rules).

To illustrate this idea, figure 5.16 shows the merging meta-model for our simple transformation. We suggest that the semantics of the transformation follows a consumption/production technique: instances of output classes are created while consuming instances of input classes. In order to keep track of the modeling elements that have been processed by the transformation, we introduce an abstract meta-class *Element* to gather modeling elements consumed by the transformation. This class introduces an attribute *name* to identify processed elements and a boolean attribute *done* to identify input elements that have been already consumed.

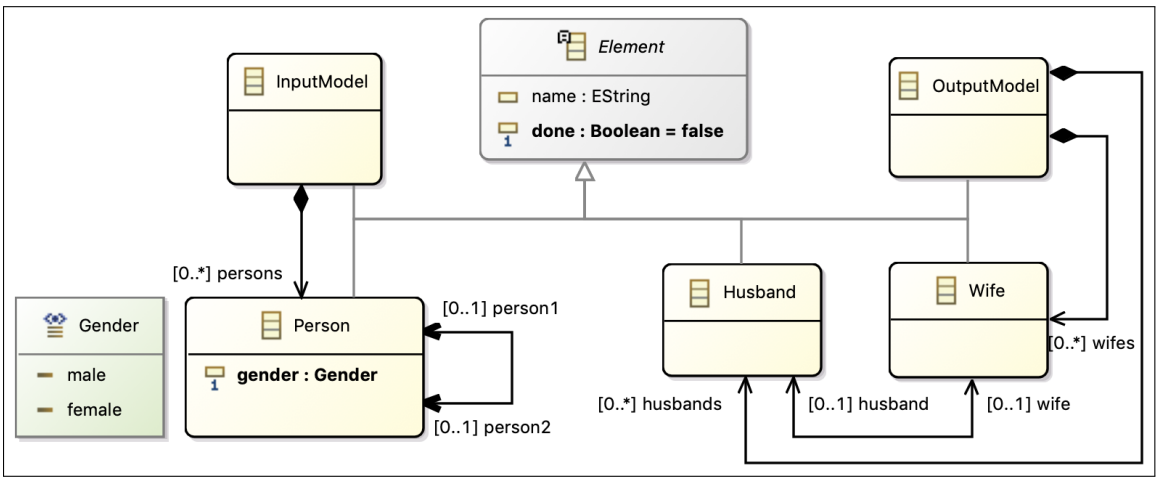

Figure 5.16: Merging meta-models

5.3.2 Step 2: generation of the *"model construction"* specification

From the merging meta-model, Meeduse automatically generates a B specifications with basic modeling operations as well as structural invariants. This technique allows one to write the transformation rules in the B language. Figure 5.17 presents the structural part of the resulting B machine. The behavioral part of the generated B machine provides all basic operations for model manipulation: getters, setters, constructors and destructors; for this reason we refer to this machine as the "model construction" machine. Figure 5.18 shows the specification of two generated operations. Operation *Husband_NEW* is a constructor that creates an instance of class *Husband*. This operation takes an element from the possible objects defined by abstract set *ELEMENT* and adds it to the set of existing instances of class *Husband*. Operation *Husband_SetWife* is a setter that straightforwardly assigns a value to the bi-directional reference husband/wife. Notice that this step is analogous to what happens in MDE tools that generate code from meta-models. For instance, from a given meta-model definition EMF can generate Java modeling code (getters, setters, etc), that can be used to program model transformation in Java. In the same way, Meeduse generates a B machine that can in turn be used to specify model transformations in B.

The B specification issued from our simple meta-model is about 335 lines of code with 34 basic operations which are proved correct (with respect to the structural invariant) by construc-

tion. Proofs were carried out using AtelierB, for this simple specification it generated 120 proof obligations, which were all automatically proved by the theorem prover. This means that the use of these modeling operations guarantees the preservation of the structural properties (invariant) of the meta-model.

MACHINE
  $simpleModel$
SETS
  $Gender = \{male, female\};$
  $ELEMENT$
ABSTRACT_VARIABLES
  $Element,$
  $InputModel,$
  $Person,$
  $OutputModel,$
  $Husband,$
  $Wife,$
  $person\_1\_2,$
  $persons,$
  $husband\_wife,$
  $wifes,$
  $husbands,$
  $Element\_done,$
  $Person\_gender$

INVARIANT
  $Element \in \mathcal{F}\ (ELEMENT) \wedge$
  $InputModel \subseteq Element \wedge$
  $Person \subseteq Element \wedge$
  $OutputModel \subseteq Element \wedge$
  $Husband \subseteq Element \wedge$
  $Wife \subseteq Element \wedge$
  $person\_1\_2 \in Person \rightarrowtail\!\!\!\!\!+ Person \wedge$
  $persons \in Person \nrightarrow InputModel \wedge$
  $husband\_wife \in Husband \rightarrowtail\!\!\!\!\!+ Wife \wedge$
  $wifes \in Wife \nrightarrow OutputModel \wedge$
  $husbands \in Husband \nrightarrow OutputModel \wedge$
  $Element\_done \in Element \nrightarrow \mathbf{BOOL} \wedge$
  $Person\_gender \in Person \rightarrow Gender$

Figure 5.17: Structural part of the modeling specification

Husband_NEW (*aHusband*) =
PRE
  $aHusband \in ELEMENT$
THEN
  $Husband := Husband \cup \{aHusband\}\ ||$
  $Element := Element \cup \{aHusband\}$
END;

Husband_SetWife (*aHusband,aWife*) =
PRE
  $aHusband \in Husband$
  $\wedge\ aWife \in Wife$
THEN
  $husband\_wife(aHusband) := aWife$
END;

Figure 5.18: Generated constructor and setter for class Husband

5.3.3 Step 3: writing the transformation rules

A model transformation is manually written in a new B machine as a set of B operations that can reuse the modeling operations defined in the *simpleModel* machine (figures 5.17 and 5.18). Each transformation rule is defined as a B operation composed of two parts: the guard and the

action. The guard gives the conditions under which the rule can be triggered, and the action specifies a sequence of calls to modeling operations (from machine *simpleModel*) whose effect is to create the output model. Figure 5.19 gives the two transformation rules that we defined for our simple example.

Input2Output =
  ANY *input* **WHERE**
    $input \in InputModel$
    $\wedge\ input \notin OutputModel$
  THEN
    OutputModel_NEW(*input*)
  END;

Person2HusbandWife =
  ANY *output*, *p1*, *p2* **WHERE**
    $output \in OutputModel$
    $\wedge\ p1 \in Person \wedge p2 \in Person$
    $\wedge\ Person\_gender(p1) = male$
    $\wedge\ Person\_gender(p2) = female$
    $\wedge\ ((p1 \mapsto p2) \in person\_1\_2 \vee (p2 \mapsto p1) \in person\_1\_2)$
    $\wedge\ Element\_done[\{p1, p2\}] = \{\textbf{FALSE}\}$
    $\wedge\ (husbands \cup wifes)[\{p1,p2\}] = \emptyset$
  THEN
    Husband_NEW(*p1*) ;
    Wife_NEW(*p2*) ;
    Husband_SetWife(*p1*, *p2*) ;
    OutputModel_AddHusbands(*output*, *p1*) ;
    OutputModel_AddWifes(*output*, *p2*) ;
    Element_SetDone(*p1*, **TRUE**) ;
    Element_SetDone(*p2*, **TRUE**)
  END
END

Figure 5.19: Transformation rules written in B

Operation *Input2Output* creates an *OutputModel* for each *InputModel*. It takes any existing instance of class *InputModel* ($input \in InputModel$) which has not been yet transformed (condition $input \notin OutputModel$) and then its action creates the new instance of *OutputModel* by calling basic operation *OutputModel_NEW(input)*.

Operation *Person2HusbandWife* takes two instances of class *Person* representing a married couple (defined by parameters $p1$ and $p2$) and translates them into instances of classes *Husband* and *Wife* in the resulting output model. The enabling conditions for this transformation rule are:

- there exists an output model ($output \in OutputModel$)

- the input instances satisfy a pattern, $p1$ is a male (*Person_gender(p1)* = *male*), $p2$ is a female (*Person_gender(p2)* = *female*) and they are married ($(p1 \mapsto p2) \in person\_1\_2 \vee (p2 \mapsto p1) \in person\_1\_2$)
- The input instances have not been already processed ($Element\_done[\{p1, p2\}] = \{\text{FALSE}\}$ and $(husbands \cup wifes)[\{p1, p2\}] = \emptyset$)

5.3.4 Step 4: animation and debugging

In Meeduse the user can load an EMF input model and injects it in the B specifications as variable valuations. After loading the model, Meeduse asks ProB to animate the initialization and then gets the initial state of the machine. Given this state, ProB computes the list of operations whose guards are satisfied and that can then be animated from the initial state. Figure 5.20 is a screen-shot of Meeduse after loading an example input model containing six persons (View ①). The state of the machine after the initialization is displayed in View ③ of figure 5.20 (called `Output State View`). The list of operations that can be triggered in the current state is shown in view ② of figure 5.20 (`the animation view`). In our case, the only operation that can be animated at the initial state is *Input2Output*, with parameter `MyModel`.

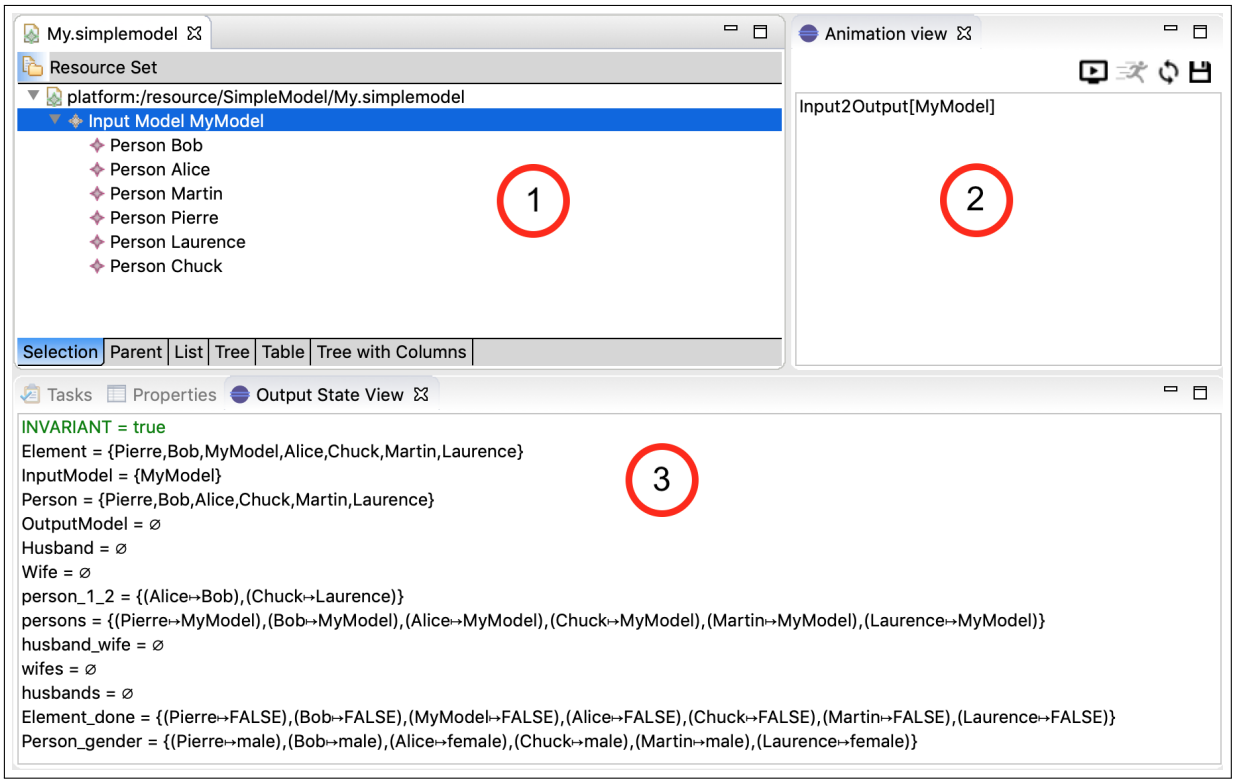

Figure 5.20: Meeduse screenshot: initial values

This interactive animation technique applies the transformation rules step-by-step to the input EMF model which is useful for debugging. The animation stops when the B specification

reaches a deadlock: a state from which there is no other possible operation to animate. It also stops when an invariant violation is detected. Figure 5.21 gives the result of the animation at the final state. In the output model, two husbands (Bob and Chuck) and two wives (Alice and Laurence) were created, with the corresponding marriage relation. The valuations of the B variables showed in the `output state view` are equivalent to the EMF model since Meeduse maintains this equivalence at every animation step.

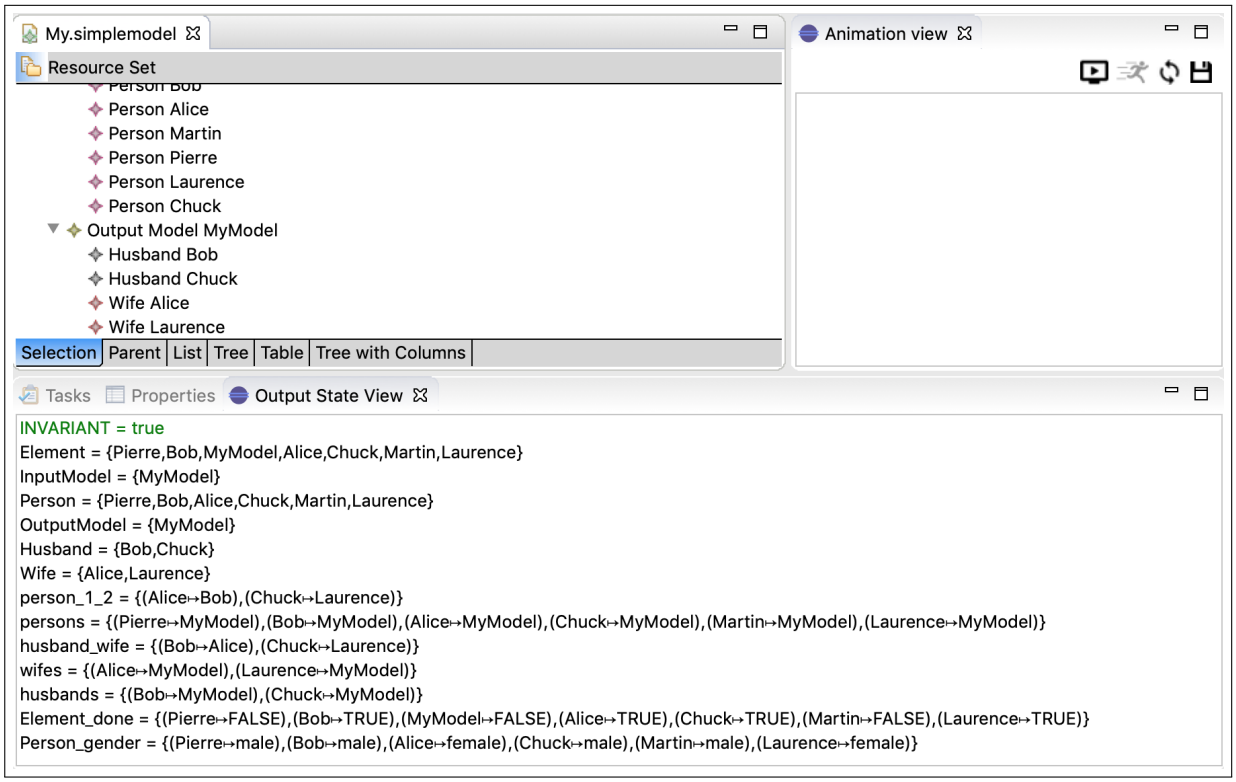

Figure 5.21: Meeduse screenshot: after animation of all rules

When the domain expert agrees with the behaviors showed by animation, transformation rules can be played without any human interaction. After loading a model the user can enable the automatic runner from the `animation view` by clicking on the corresponding icon. This runner executes a random animation: at every step it chooses randomly an operation from those provided by ProB and automatically animates it until reaching the ending state where a deadlock or an invariant violation is detected.

5.3.5 Step 5: Proving the transformation

Application of a formal method to model transformation brings several benefits to this field. Indeed, since Meeduse produces a formal specification and automatically manages the traceability between EMF models and the B machine valuations, we can go a step further towards the usage of automatic reasoning tools like model-checkers.

The invariants discussed in step 2 define the properties of our meta-model, not those of the transformation. One way to analyze the transformation and have some confidence about its correctness is to define unwanted states and ask ProB to find them by model-checking. In the following we present some example goals that we defined for our simple transformation:

$GOAL1 == \exists\, pp \,.\, (pp \in Husband \wedge Person\_gender(pp) = female)$;
$GOAL2 == \exists\, pp \,.\, (pp \in Wife \wedge Person\_gender(pp) = male)$;
$GOAL3 == \exists\, (p1, p2).((p1 \mapsto p2) \in husband\_wife$
$\quad \wedge \{p1,p2\} \not\subseteq \mathbf{dom}(person\_1\_2) \cup \mathbf{ran}(person\_1\_2))$;
$GOAL4 == Husband \cap Wife \neq \emptyset$

The three first goals are linkage properties between the input and the output meta-model. Goal1 and Goal2 for example state that an instance of class Husband is created but from a Person whose gender is female and vice-versa. Goal3 means that a husband and his wife in the output model are created but without any existing marriage link between the input persons from which they originate. Goal4 represents a forbidden property of the output meta-model and means that someone is husband and wife at same time.

Given the B specification extracted from the initial model (that of figure 5.20), we can ask ProB, from outside Meeduse, to find by model-checking states where one of these goals are satisfied. The answer of ProB is that all state space is explored without finding any of the four goals. Since the state space is entirely bounded thanks to valuations, ProB is able to compute all reachable states. This model-checking proof gives a good confidence about the correctness of the transformation. It can be applied to bigger examples in the limits of space memory and the model-checker capacities.

5.3.6 Discussion

We can remark that the chosen style for specifying the transformation rules in B reminds transformation languages available in the MDE community. The specification of the rule guard (clause **ANY**) is similar to some declarative transformation languages (it looks like the *where* condition and *checkonly* patterns in QVT relational for example). Nonetheless, the action part has a more imperative style. As B is not a specialized language for model transformation, some aspects have to be taken care explicitly, for instance we have to check that a rule is not applied several times for the same input. An important aspect that is worth mentioning is that we do not specify explicitly the execution order of the rules. The semantics of a B machine is that, at any given point during the execution, the system considers all enabled operations and makes a non-deterministic choice. The choice of the parameters in the **ANY** clause is also non-deterministic, meaning that at any execution state, the system will select any objects that satisfy the condition and use them as arguments for the operation. However, in this example we have indirectly prescribed an order of execution, because in the guard of the `Person2HusbandWife` rule we check for the existence of an object created by the `Input2Output` rule. This strategy is

also similar to some declarative transformation languages that use traceability information to infer an execution order. The dynamics of a B machine execution will be further explored in the following sections.

A final point concerns the correctness of the transformation rules. As mentioned in the previous section, the individual model construction operations (constructors, setters, ...) were proved correct, then the result of executing a sequence of operations in the action part of a rule will obviously preserve the model structural properties. However, we also need to prove that the order of the sequence of calls is correct, meaning that the preconditions of every operation in the sequence are satisfied. Lets's consider for example operation `Husband_SetWife` which can be applied only on existing instances of classes Husband and Wife: $aHusband \in Husband \wedge aWife \in Wife$. As actions `Husband_NEW` and `Wife_NEW` produce these instances, then the proof of correctness associated to the call of `Husband_SetWife` in rule `Person2HusbandWife` succeeds. For our example rules, the AtelierB generated 13 additional proof obligations, which were automatically proved. This means that we don't need to test the validity of the input models or verify the output model using the EMF validator.

5.3.7 The TTC'2019 Case Study

The call for solutions of TTC'2019 was about the transformation of Truth Tables (TT) into Binary Decision Diagrams (BDD). Among the seven participants, Meeduse was the only attempt that addressed V&V; the other solutions addressed flexibility, performance and optimality. Figure 5.22 shows the various views of Meeduse that represent our solution.

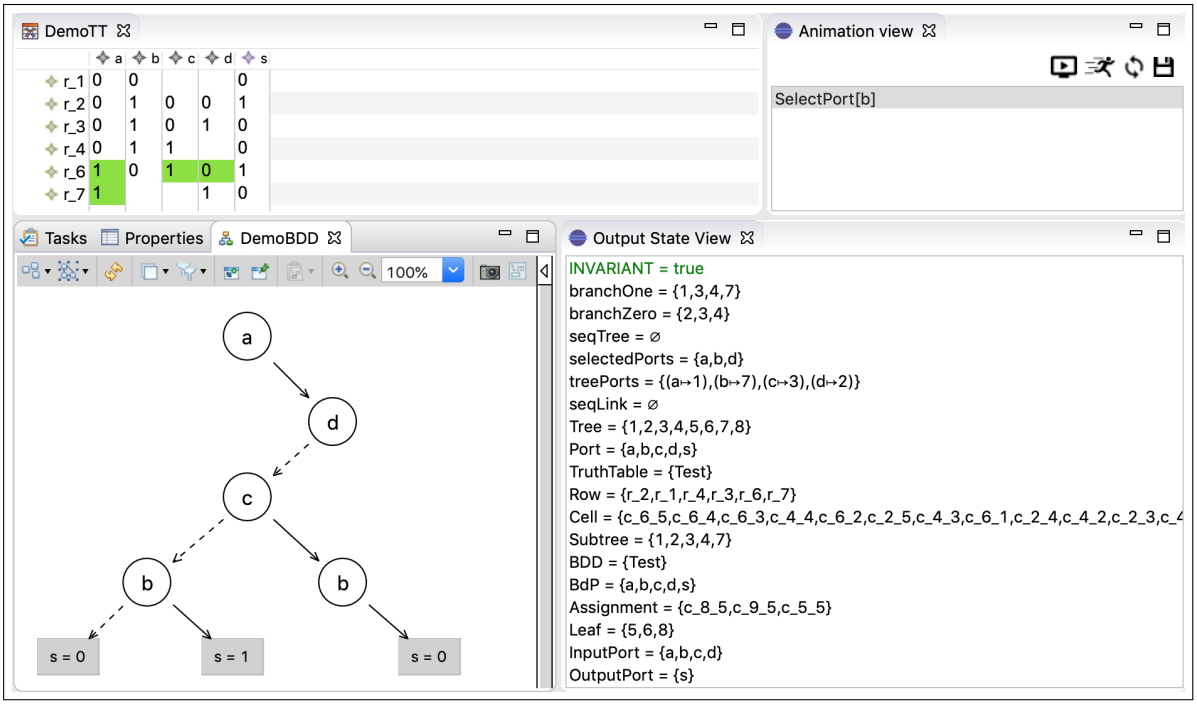

Figure 5.22: Application of Meeduse to DSL transformation

The B machine extracted from the meta-model is about 1162 lines of code from which the AtelierB generated, and automatically proved 260 proof obligations. This proof gives the guarantee that the model's properties are preserved during the DSL transformation. The operational semantics were introduced using an additional B machine whose length is about 150 lines of code. This is more concise (but may be more difficult to understand) than a reference ATL solution which is about 340 lines of ATL code. The proposed operational semantics of this case study were defined through five B operations and several additional invariants. For this case study the correctness of the operational semantics was ensured by model-checking, rather than by theorem proving because on the one hand it is less time consuming, and on the other hand, it deals with bounded state spaces that can be exhaustively checked by the ProB model-checker.

From a methodological point of view we were able to define how formal executable DSLs can be applied to define model transformations. Merging meta-models required the implementation of an additional driver, but we believe that Meeduse can be adapted to deal with two or several heterogeneous meta-models. This issue is now considered as a possible evolution of the tool. In general, we are satisfied by the application of Meeduse to the model-to-model transformation problem because as far as we know none of the existing works combine theorem proving and model-checking in a publicly available tool and which is well integrated within EMF-based platforms.

Regarding performance, it mainly depends on the performance of ProB. Execution times spent by the tool to generate the output models are given in table 5.1. The number of model elements grow exponentially. For 14 input ports and 2 output ports, Meeduse reached an out of memory. For bigger examples, it should be useful to try the experiments on a machine with higher performances than that on which we have done these measures.

Input Model	rows	input ports	output ports	Cells	Exec. Time
GeneratedI4O2Seed42	16	4	2	96	3s
GeneratedI5O2Seed5	32	5	2	224	5s
GeneratedI8O2Seed68	256	8	2	2560	1mn3s
GeneratedI8O4Seed68	256	8	4	3072	2mn1s
GeneratedI10O2Seed68	1024	10	2	20480	18mn8s
GeneratedI12O2Seed7634	4096	12	2	57344	6h40mn

Table 5.1: Some performance measures

For readability, we believe that the verbose notation of the B method is accessible because it recalls some programmatic styles. It is often said to be less difficult than other formal notations. Our transformation file is about 150 lines which remains reasonable. However, the B specification that we provide is expected to be readable for a formal methods expert, may be more readable than an ATL or a QVT transformation. But this intuition needs some empirical studies in order to be confirmed.

Conclusion and perspectives

« *Maintenant l'abstrait s'était matérialisé,*
l'être enfin compris avait aussitôt perdu de son pouvoir de rester invisible, […]
tout ce qui avait paru, jusque-là, incohérent à mon esprit,
devenait intelligible, se montrait évident, comme une phrase, n'offrant aucun
sens tant qu'elle reste décomposée en lettres disposées au hasard, exprime,
si les caractères se trouvent replacés dans l'ordre qu'il faut,
une pensée que l'on ne pourra plus oublier. »

Marcel Proust

This document presented my research works during the last fourteen years. I addressed two topics: Model-Driven Security and Domain-Specific Languages. The guiding principle of my contributions is the combination of two well-known paradigms: Formal Methods (FM) and Model-Driven Engineering (MDE); to which I refer using acronym FMDE. During this period I had the opportunity to (co-)supervise many students for their PhD, M2 or final study engineer projects. I also collaborated with many colleagues and have been member of various research projects. My works led to the development of two tools: B4MSecure (Idani, A. and Ledru, 2015) and Meeduse (Idani, A., 2020b). These tools reached a good level of maturity and are no longer simple prototypes. They have been applied on several realistic case studies and showed by practice their strengths. B4MSecure is currently used in the "Information Security" lecture of the M2 MoSIG at Grenoble. It is exploited in a lab that introduces students to the design of Access Control policies, and also to testing via animation and to the insider threat problem. Regarding Meeduse, it is today the only existing language workbench (LWB) that favours both formal reasoning − via theorem proving − and the execution of the DSL − via animation and model-checking.

Several research directions that are mentioned in the introduction have not been discussed in details in this document, for space reasons and also because some of them are not yet published. I'm still actively working on these topics, with several perspectives.

Formal MDS

Nowadays, information technology architectures support interconnection between systems and favour a greater interoperability and a centralized access to data. However, despite these advantages, they expose information systems to a multitude of security risks. Indeed, organisations manage sensitive information and security vulnerabilities can lead to financial loss, dilapidation of the brand image, shutdown of production systems or leakage of confidential information. Hence, information flows must be controlled to adequately ensure information security and to enforce the ACIT properties (Availability, Confidentiality, Integrity, Traceability) of Secure Information Systems. Separation of concerns and abstraction/refinement techniques have been proposed to master the complexity of these systems. First, separation of concerns distinguishes functional aspects of the IS (data model and the associated business logic) from security concerns. Second, it appears that security can be considered at several abstraction levels: from high level ACIT properties, close to the concerns of users, to security techniques and mechanisms of the hardware and software platform.

Significant efforts have been dedicated to the careful definition of security policies and their deployment on a given technology. The Model-Driven Security (MDS) (Basin et al., 2011) approach addresses this challenge by trying to define the relationship between high-level security modeling, expressed in terms of permission rules, and the implementation level. Platform Independent Models (PIM) promote a real separation of concerns between the functional requirements and the security policy. MDS makes the link, using transformation tools, between the PIM and the Platform Specific Model (PSM), where the PSM is a more concrete model that can be mapped to a given technology platform (programming language, operating system, ...). However, validation of the various models remains error prone without the assistance of formal analysis tools and consequently the encoded security policy is poorly understood and difficult to maintain, correct or adapt. Current validation activities are limited to structural aspects of the models, where functional and security models are validated separately. However, evolutions of the functional state of an IS may change the context of several permissions, which may lead to security threats. The validation activities thus require tools that can take into account dynamic aspects of **both functional and security aspects**. These tools can come in the form of model checkers and animation tools, dedicated to formal specification techniques.

My works in this field propose a way to guarantee a **security-by-design** approach in which the MDS benefits from an automated formal support based on the formal B method. Indeed, formal B specifications take into account behavioural aspects of the secure IS and deal with interactions between the various models thanks to its composition mechanism. The formal analysis techniques addressed are intended to exhibit interesting behaviours; for example,

- to check liveness properties of the IS, as security constraints can be too strict and then block the system;
- to check that the access control forbids the system to enter undesirable functional states;

- to find malicious behaviours like insider threats performed by authorized users in order to bypass the security policies;
- to produce positive and negative test scenarios, to check the correctness of normal behaviours and the effects produced by non-nominal (undesired) variations of these normal scenarios;

A backward symbolic search algorithm has been proposed in the PhD thesis of A. Radhouani in order to extract attack scenarios that are made possible by the evolution of the functional state of the IS. In addition, this approach is able to extract malicious behaviours performed by a single user or several users on coalition. The technique presented in Chapter 3 has been complemented with a two-steps approach: the first step focuses on the functional model regardless of users and their roles in order to extract the sequences of functional operations that reach the malicious goal; and the second step guides ProB with a CSP model to identify users able to perform such attacks according to the security model. This work led to the development of GenISIS, a tool that is used to exhibit execution paths from a B modelling of an IS. GenISIS has been experimented with several case studies such as the meeting scheduler example discussed in (Basin et al., 2009, Radhouani et al., 2015), the medical IS studied in (Ledru et al., 2015b) and the conference review IS inspired by(Zhang et al., 2008). For each example, GeneISIS aimed to reach the same malicious goal as handled in the article which addressed the same example, and it was able to extract all reported attacks. Some metrics about these experiments are given in Table 1.

Case study	Operations	Variables	Permissions	Roles	Users	scenarios
Library	13	4	3	2	3	8
Medical IS (Ledru et al., 2015b)	15	9	3	4	3	10
Meeting scheduler (Basin et al., 2009)	23	7	5	3	3	8
Bank IS (Bandara et al., 2010)	31	11	4	2	3	9
Conference Review (Zhang et al., 2008)	48	24	8	3	4	14

Table 1: Summary table of experiments

My works mainly addressed the modeling activities and were based on Platform Independent Models (PIM), described using UML models and their associated formal B specifications. However, in addition to modeling notions, MDS also promotes the transformation of the PIM into a PSM. The objective is to translate access control into concrete security mechanisms of a target infrastructure. In practice, this transformation usually includes manual coding activities. The challenge is therefore to guarantee that security models, graphically designed and formally

validated, correspond to a deployed security policy. One way to address this perspective is to apply conformance testing.

From the PIM level, testing activities can take several directions: (1) validation of the normal behavior of the IS regarding its security rules, (2) automated generation of security tests from valid functional scenarios, etc. The PSM level validation can be done by verifying whether the PSM complies with the specification and its validation activities. In this context the goal is twofold: (1) extract PSM models of target technologies like web applications and databases, and (2) translate the validation activities carried out on PIM models into validation activities at the implementation level. This can be achieved by the extraction of security monitors, and/or the production of log files that can be further analysed applying trace analysis techniques.

Formal xDSLs

Several research works have been devoted by the MDE community in order to deal with executable DSLs. Combemale (2015) state that the intention is to support early validation and verification in the development process. Indeed, an executable model not only represents the structural features of a system, but also deals with its behaviour. The model is therefore intended to behave like the final system should run, which provides the capability to simulate, animate and debug the system's properties before its implementation. In the literature there are two major approaches to implement the execution semantics of a DSL: translational approaches and in-place approaches. The former translate the DSL semantics into a well-established semantic domain that is assisted by numerous tools. The latter weave the execution semantics into meta-models, which is often done by extending the semantic domain of a DSL with action languages. Every approach has its strengths and limitations that can be summarized by:

- Translational approaches:
  - *Strengths*: apply available tools, such as animators and/or model-checkers to ensure the execution capabilities of the DSL.
  - *Limitations*: first, often these approaches require complex transformations to implement the mapping from the DSL to the target semantic domain, and second, the execution results are only obtained in the target domain.
- In-place approaches:
  - *Strengths*: allow a more intuitive definition of executable DSLs since the language engineer has only to deal with concepts of the DSL and not with another target language.
  - *Limitations*: require to implement for each DSL all the execution-based tools.

Since the objective of an xDSL is to ensure early validation during the development process, the developer must have some confidence in the underlying verification tool. Nonetheless, on

the one hand, Kosar et al. (2016) and Iung et al. (2020) show that the verification feature of existing language workbenches (LWBs) is very limited; and on the other hand, the dependability of the resulting system is strongly related to the correctness of the DSL. Voelter et al. (2019) analysed the risks of using LWBs regarding the introduction of faults into a critical software component. The authors observed that existing LWBs have not been developed using a safety process and attested that "*particular DSLs could be, but that is only of limited use if the underlying LWB is not*". In this sense, a translational approach is much more pragmatic because it reuses well-established verification tools, that are often widely accepted by the formal methods community.

In my opinion, bridging the gap between both worlds (xDSLs and FM) does not require innovative solutions and can be done by integrating well-established tools provided by both the Formal Methods community and the MDE community. The lack of automated formal reasoning in LWBs is not due to the complexity of formal methods and their mathematical background, but originates from the lack of initiatives that are dedicated to the integration of both techniques. A supporting argument could be the assertion by Kosar et al. (2016) that "*researchers within the DSL community are more interested in creating new techniques than they are in performing rigorous [empirical] evaluations*". Unfortunately, not only the integration attempts remain poor, but also the applications of existing approaches remain limited to illustrative examples without going further towards realistic safety-critical requirements. This observation may explain why existing approaches (Rivera et al., 2009, Gargantini et al., 2010b, Merilinna and Pärssinen, 2010, Tikhonova, 2017b, Zalila et al., 2013) are not discussed at all in the systematic mapping studies of Kosar et al. (2016) and Iung et al. (2020). Meeduse is a LWB providing solutions to this limitation. It allows one to formally define and reason about the semantics of xDSLs using the formal B method. The underlying approach is a translational approach, but it goes a step further in comparison with existing translational approaches by providing effective solutions to the limitations mentioned above. The applications and case studies presented in Chapter 5 show that the integration of executable DSLs together with theorem proving, animation and model-checking is viable and should be explored further.

The usage of B in my works appears as a good choice for two reasons: (1) the availability of a rich UML-to-B state of the art, which provides a viable translation from ECore into B (considering that ECore is conformant to the MOF, which is a restriction of UML); and (2) the usage of theorem proving, in addition to model-checking, to guarantee zero-fault DSLs. In most application domains such as Requirements Engineering, Enterprise Architectures, Business Process Management and Legal Contracts, where DSLs do exist, the commonly used verification tools are model checkers and/or SAT/SMT solvers. Meeduse introduces theorem proving to these domains. Besides my own experience and judgement, the B Method has been compared to other state based formal methods and tools in (Mashkoor et al., 2018) and got several good points. Regarding scalability, which is "*the ability to be well applicable to arbitrarily large and complex projects*", the B method is ranked (Good). The work also highly ranks the verification features of B and its tool support. Figure 1 gives the evolution time-line of the tool with the

major highlights and current and short-term perspectives.

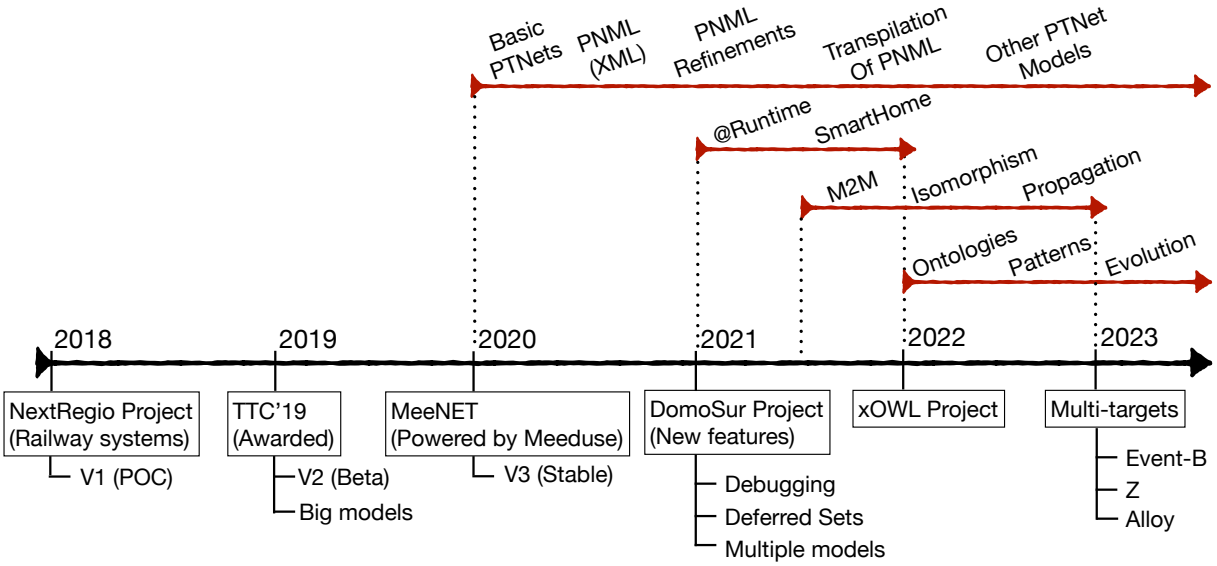

Figure 1: Evolution time-line of Meeduse

The red time-lines refer to DSL tools that are powered by Meeduse and which have their own existence, such as MeeNET. The latter investigates several interesting research directions such as DSL refinements and transpilation. Meeduse has been experimented on various kinds of applications in order to evaluate its strength. For example, in the smart-home domain (project DomoSur) the objective was the execution of the DSL at run-time, being inspired by (Körner et al., 2020). The approach also covers model transformations (M2M) and currently, I am working on bi-directional transformations by addressing two features: (1) proving the isomorphism of a transformation in B, and (2) updating the input model from changes done in the output model (propagation).

The xOWL project (executable OWL), started last year, addresses knowledge engineering. It has a different view and studies the possibility to consider other application domains, which would allow the analysis of the implications of FMDE outside safety-critical systems. The motivation of the project is that domain ontologies evolve continuously, which leads to several problems, especially change impact analysis and resolution. Among existing works, pattern-driven techniques have been proposed to provide guidance during the ontology evolution so that it remains consistent. In the xOWL project, ontologies are described via the Ontology Web Language (OWL). Considering that OWL is a DSL, the proposal is to rethink the underlying evolution patterns by means of execution semantics that apply the expected changes to a given ontology. The W3C functional syntax of OWL has been instrumented in Meeduse, leading to a lightweight development approach. Indeed, the development effort is limited to the specification of the evolution patterns and their verification and validation. All the other features of xOWL (execution, verification and debugging) are provided by Meeduse.

Personal opinion

Undoubtedly, the usage of a formal method with well-established verification tools is a solution to neutralize bugs that may originate from a modeling language. The question is: "*how to provide the good mixture between correctness and expressiveness?*". I believe that the answer depends on the application domain of the language. Indeed, formal methods are often negatively perceived by developers due to their mathematical notations, and consequently translational approaches have a limited usage for general purpose applications. Nonetheless, formal methods showed their strengths in the safety-critical domain, and they became a strong requirement to ensure zero-fault applications. On the other side, MDE provides several benefits to this field due to the usage of DSLs and the ability to share, visualize and communicate about domain concepts. Both formal methods and model-driven engineering are desirable in safety critical systems because domain-specific representations are omnipresent, as well as the use of provers and model-checkers. I assume that the reader may agree with this claim, even if it appears that for larger scale projects formal methods are not as widespread, because of the overhead they may create during the development activities.

Note that The backbone of my works, for MDS and DSLs, is the UML-to-B translation. It can be considered that MDS is a particular case of DSLs and hence an interesting technical perspective would be to revisit B4MSecure using Meeduse. In fact, B4MSecure translates UML and RBAC into B applying model-to-model transformations written in Java. Considering that UML, RBAC and B are DSLs, and having their meta-models, it is possible to rewrite the transformations using B, prove their correctness, and execute them in Meeduse. Even if Meeduse is a recent tool (its reference papers appeared in 2020), the underlying approach is not new since it provides a translation semantics to a DSL. The limitations of this kind of approach were widely discussed in the literature (Bryant et al., 2011). The first one is that it "*is very challenging to correctly map the constructs of the DSL into the constructs of the target language*". The use of the B method in Meeduse provides objective answers to this observation because the semantics of meta-models is defined by means of OCL constructs in the MOF. These are built on the set theory and the first order predicate logic, which are also the foundations of the B language. Structurally a meta-model is a restriction of UML, which is comforting because this means that mapping a DSL into B is not very challenging. It can be done via a classical UML-to-B approach, which has been addressed in the past (since the 2000s) by a plethora of techniques. My research works made this translation more effective than existing techniques; first, by providing tools and second by embedding ProB in these tools.

Furthermore, the second limitation of translational approaches, as discussed in (Bryant et al., 2011), is that "*the mapping of execution results back into the DSL is not covered*". Indeed, in the existing works the V&V activities are only obtained in the target domain because getting back the results in the source language is often claimed to be difficult and requires to extend the abstract syntax of the input DSL in order to model these results. Existing works start from a model (in UML or in a given DSL), produce a formal specification and then they "get lost"

in the formal process. By embedding ProB, Meeduse gave a new vision to the integration of formal methods and modeling languages.

Every UML-to-B approach has its advantages and limitations. Obviously for a better coverage of UML a combination of the various approaches is needed. The major challenge was how to gather into a unifying framework several UML-to-B approaches from the rich state of the art. In B4MSecure and Meeduse the user is able to select the desired UML-to-B transformation technique and also to combine rules coming from various techniques. This is interesting because UML has been mapped into numerous state-based formal languages with similar principles (*e.g.* Z, Object-Z, etc), and therefore the only remaining piece is to reuse these mappings within the tool. I am aware that in order to broaden the spectrum of B4MSecure and Meeduse several target formal approaches have to be addressed. This objective is reachable since the power of both tools comes from ProB, and ProB is itself a multi-target platform covering (in addition to B) Event-B, Z, Alloy and TLA+.

Bibliography

Abrial, J.-R. (1996). *The B-book: assigning programs to meanings*. Cambridge University Press.

Abrial, J.-R. and Mussat, L. (1998). Introducing dynamic constraints in B. In *B'98: Recent Advances in the Development and Use of the B Method, Second International B Conference, Montpellier, France, April 22-24, 1998, Proceedings*, pages 83–128. LNCS.

Ahn, G.-J. and Hu, H. (2007). Towards realizing a formal rbac model in real systems. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies (SACMAT'07)*, pages 215–224, New York, NY, USA. Association for Computing Machinery.

Aït-Ameur, Y., Ait-Sadoune, I., Hacid, K., and Oussaid, L. M. (2017). Formal modelling of ontologies within Event-B. In *First International Workshop on Handling IMPlicit and EXplicit knowledge in formal system development*. https://hal.archives-ouvertes.fr/hal-01636944/file/IMPEX_2017_paper_3.pdf.

Andova, S., van den Brand, M., Engelen, L., and Verhoeff, T. (2012). MDE basics with a DSL focus. In *Formal Methods for Model-Driven Engineering - 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*, volume 7320 of *LNCS*, pages 21–57. Springer.

Attiogbe, C. (2009). Semantic Embedding of Petri Nets into Event-B. In *Integration of Model-based Formal Methods Tools (IM_FMT @ IFM'2009)*, Dusseldorf, Germany. http://www.lina.sciences.univ-nantes.fr/apcb/IM_FMT2009/index.html.

Baar, T. (2005). Non-deterministic constructs in OCL - what does any() mean. In *Model Driven - 12th International SDL Forum*, volume 3530 of *LNCS*, pages 32–46. Springer.

Bandara, A., Shinpei, H., Jurjens, J., Kaiya, H., Kubo, A., Laney, R., Mouratidis, H., Nhlabatsi, A., Nuseibeh, B., Tahara, Y., Tun, T., Washizaki, H., Yoshioka, N., and Yu, Y. (2010). *Security Patterns: Comparing Modeling Approaches*. IGI Global.

Basin, D., Clavel, M., Doser, J., and Egea, M. (2009). Automated analysis of security-design models. *Information & Software Technology*, 51.

Basin, D., Clavel, M., and Egea, M. (2011). A Decade of Model-Driven Security. In *Proceedings of the 16th ACM Symposium on Access Control Models and Technologies*, SACMAT'11, pages 1–10. ACM.

Basin, D., Doser, J., and Lodderstedt, T. (2006). Model driven security: From UML models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.*, 15(1).

Becker, M. Y. and Nanz, S. (2010). A logic for state-modifying authorization policies. *ACM Trans. Inf. Syst. Secur.*, 13(3):20:1–20:28.

Bert, D., Potet, M.-L., and Stouls, N. (2005). GeneSyst: a Tool to Reason about Behavioral Aspects of B Event Specifications. Application to Security Properties. In H. Treharne, S. King, M. C. Henson, and S. A. Schneider, editors, *ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users*, volume 3455 of *LNCS*. Springer-Verlag.

Bjørner, D. (2010). Rôle of domain engineering in software development. In *Perspectives of Systems Informatics*, pages 2–34. Springer.

Bousse, E., Leroy, D., Combemale, B., Wimmer, M., and Baudry, B. (2018). Omniscient debugging for executable dsls. *Journal of Systems and Software*, 137:261–288.

Bryant, B. R., Gray, J., Mernik, M., Clarke, P. J., France, R. B., and Karsai, G. (2011). Challenges and directions in formalizing the semantics of modeling languages. *Comput. Sci. Inf. Syst.*, 8(2):225–253.

Butler, M. J., Körner, P., Krings, S., Lecomte, T., Leuschel, M., Mejia, L., and Voisin, L. (2020). The first twenty-five years of industrial use of the b-method. In ter Beek, M. H. and Nickovic, D., editors, *Formal Methods for Industrial Critical Systems - 25th International Conference, FMICS 2020, Vienna, Austria, September 2-3, 2020, Proceedings*, volume 12327 of *Lecture Notes in Computer Science*, pages 189–209. Springer.

Butler, M. J. and Leuschel, M. (2005). Combining CSP and B for specification and property verification. In *International Symposium of Formal Methods - FM 2005*, volume 3582 of *Lecture Notes in Computer Science*, pages 221–236. Springer.

Butler, M. J., Raschke, A., Hoang, T. S., and Reichl, K. (2018). *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 6th Int. Conference, ABZ 2018*, volume 10817 of *LNCS*. Springer.

Combemale, B. (2015). *Towards Language-Oriented Modeling*. Habilitation à diriger des recherches, Université de Rennes 1.

Cuadrado, J. S. and Molina, J. G. (2007). A phasing mechanism for model transformation languages. In *SAC*, pages 1020–1024.

Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646.

Deantoni, J. (2016). Modeling the behavioral semantics of heterogeneous languages and their coordination. In *2016 Architecture-Centric Virtual Integration (ACVI)*, pages 12–18.

Dijkstra, E. W. (1975). Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communication of the ACM*, 18(8):453–457.

Eclipse (2012). Acceleo.

Eclipse (2021). Eclipse QVT Operational 2021-12 (3.10.5). https://projects.eclipse.org/projects/modeling.mmt.qvt-oml. Accessed: 20-12-2021.

Efftinge, S. (2006). openarchitectureware 4.1 expressions framework reference.

ePNK Homepage, T. (2020). http://www2.compute.dtu.dk/~ekki/projects/ePNK/index.shtml. Accessed: 15-12-2020.

EProvide (2020). http://eprovide.sourceforge.net. Accessed: 15-12-2020.

ERA, UNISIG, E. E. U. G. (2016). System Requirements Specification, SUBSET-026. Technical report, European Railway Agency. Version 3.6.0.

Essamé, D. (2004). La méthode B et l'ingénierie Système. *TSI (Technique et Science Informatiques)*, 23(7):929–938.

Ferraiolo, D. F., Sandhu, R. S., Gavrila, S. I., Kuhn, D. R., and Chandramouli, R. (2001). Proposed NIST standard for Role-based Access Control. In *ACM Transactions on Information and System Security (TISSEC'-01)*, pages 224–274.

Fisler, K., Krishnamurthi, S., Meyerovich, L. A., and Tschantz, M. C. (2005). Verification and change-impact analysis of access-control policies. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 196–205, New York, NY, USA. ACM.

Floyd, R. W. (1993). *Assigning Meanings to Programs*. Springer Netherlands, Dordrecht.

Fowler, M. (2005). Language workbenches: The killer-app for domain specific languages? http://martinfowler.com/articles/languageWorkbench.html.

Garcia-Dominguez, A. and Hinkel, G. (2019). Truth Tables to Binary Decision Diagrams. In Garcia-Dominguez, A., Hinkel, G., and Krikava, F., editors, *Proceedings of the 12th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2019) federation of conferences*, CEUR Workshop Proceedings. CEUR-WS.org.

Gardner, T., Griffin, C., Koehler, J., and Hauser, R. (2003). A review of OMG MOF 2.0 Query / Views / Transformations submissions and recommendations towards the final standard. Meta-Modelling for MDA Workshop.

Gargantini, A., Riccobene, E., and Scandurra, P. (2010a). Combining formal methods and mde techniques for model-driven system design and analysis. *Advances in Software*, 3(1&2).

Gargantini, A., Riccobene, E., and Scandurra, P. (2010b). Combining formal methods and mde techniques for model-driven system design and analysis. *International Journal On Advances in Software*, 1.

Gehlot, V. and Nigro, C. (2010). An introduction to systems modeling and simulation with colored petri nets. In *Proceedings of the 2010 Winter Simulation Conference, WSC 2010, USA, 5-8 December 2010*, pages 104–118.

Gogolla, M., Büttner, F., and Richters, M. (2007). Use: A uml-based specification environment for validating uml and ocl. *Science of Computer Programming*, 69(1):27–34. Special issue on Experimental Software and Toolkits.

Greitzer, F. L. (2019). Insider Threats: It's the HUMAN, Stupid! In *Proceedings of the Northwest Cybersecurity Symposium*, NCS '19, New York, NY, USA. Association for Computing Machinery.

Hagalisletto, A. M., Bjørk, J., Yu, I. C., and Enger, P. (2007). Constructing and refining large-scale railway models represented by petri nets. *IEEE Trans. Systems, Man, and Cybernetics, Part C*, 37(4):444–460.

Harel, D. and Rumpe, B. (2004). Meaningful modeling: What's the semantics of "semantics"? *Computer*, 37:64 – 72.

Hartmann, T. and Sadilek, D. A. (2008). Undoing operational steps of domain-specific modeling languages. In *Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling (DSM'08) - University of Alabama at Birmingham*.

Hazem, L., Levy, N., and Marcano-Kamenoff, R. (2004). UML2B : un outil pour la génération de modèles formels. In Julliand, J., editor, *AFADL'2004 - Session Outils*.

Hillah, L. M., Kindler, E., Kordon, F., Petrucci, L., and Trèves, N. (2009). A primer on the Petri Net Markup Language and ISO/IEC 15909-2. *Petri Net Newsletter*, 76:9–28.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communication of the ACM*, 12(10):576–580.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Höhn, S. and Jürjens, J. (2008). Rubacon: Automated support for model-based compliance engineering. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE'08, pages 875–878, New York, NY, USA. Association for Computing Machinery.

Homepage, P. F. (2020). https://pnml.lip6.fr. Accessed: 15-12-2020.

Homoliak, I., Toffalini, F., Guarnizo, J., Elovici, Y., and Ochoa, M. (2019). Insight Into Insiders and IT: A Survey of Insider Threat Taxonomies, Analysis, Modeling, and Countermeasures. *ACM Computing Surveys*, 52(2).

Infrabel (2017). Stabilisation du nombre de dépassements de signaux sur le rail en 2016. https://www.infrabel.be/fr/presse/stabilisation-du-nombre-depassements-signaux-rail-2016.

International Union of Railways (UIC) (2016). RailTopoModel - Railway infrastructure topological model. ISBN 978-2-7461-2513-1.

ISO/IEC (2011). *Systems and software engineering – High-level Petri nets – Part 2: Transfer format, International Standard ISO/IEC 15909-2*.

Iung, A., Carbonell, J., Marchezan, L., Rodrigues, E. M., Bernardino, M., Basso, F. P., and Medeiros, B. (2020). Systematic mapping study on domain-specific language development tools. *Empirical Software Engineering*, 25(5):4205–4249.

Idani, A. (2006). *B/UML : Mise en relation de spécifications B et de descriptions UML pour l'aide à la validation externe de développements formels en B*. PhD thesis, Université de Grenoble 1.

Idani, A. (2009). UML models engineering from static and dynamic aspects of formal specifications. In *14th International Conference Exploring Modeling Methods for Systems Analysis and Development (EMMSAD), held at CAiSE 2009*, volume 29 of *LNBIP*, pages 237–250. Springer.

Idani, A. (2020a). Dependability of Model-Driven Executable DSLs, Critical Review and Solutions. In *14th European Conference on Software Architecture, Companion Proceedings*, volume 1269 of *CCIS*, pages 358–373. Springer.

Idani, A. (2020b). Meeduse: A tool to build and run proved DSLs. In Dongol, B. and Troubitsyna, E., editors, *16th International Conference on Integrated Formal Methods (IFM)*, volume 12546 of *LNCS*, pages 349–367. Springer.

Idani, A. (2021). A Lightweight Development of Outbreak Prevention Strategies Built on Formal Methods and xDSLs. In *2nd ACM European Symposium on Software Engineering (ESSE)*. ACM.

Idani, A. (2022). Formal model-driven executable DSLs. *International Journal on Innovations in Systems and Software Engineering (ISSE)*, 18(1).

Idani, A., Boulanger, J.-L., and Philippe, L. (2009). Linking paradigms in safety critical systems. *International Journal of Computers and their Applications (IJCA), Special Issue on the Application of Computer Technology to Public Safety and Law Enforcement*, 16(2).

Idani, A. and Coulette, B. (2008). Towards reverse-engineering of UML views from structured formal developments. In Cordeiro, J. and Filipe, J., editors, *Proceedings of the Tenth International Conference on Enterprise Information Systems (ICEIS)*, pages 94–103.

Idani, A., Labiadh, M., and Ledru, Y. (2010a). Infrastructure dirigée par les modèles pour une intégration adaptable et évolutive de UML et B. *Ingénierie des Systèmes d'Information*, 15(3):87–112. Extended version.

Idani, A., Labiadh, M., and Ledru, Y. (2010b). Infrastructure dirigée par les modèles pour une intégration adaptable et évolutive de UML et B. *Ingénierie des Systèmes d'Information*, 15(3):87–112. Extended version.

Idani, A. and Ledru, Y. (2015). B for Modeling Secure Information Systems, The B4MSecure Platform. In Butler, M. J., Conchon, S., and Zaidi, F., editors, *17th International Conference on Formal Engineering Methods (ICFEM), Paris, France, November 3-5*, volume 9407 of *LNCS*, pages 312–318. Springer.

Idani, A., Ledru, Y., Ait-Wakrime, A., Ben-Ayed, R., and Bon, P. (2019a). Towards a Tool-Based Domain Specific Approach for Railway Systems Modeling and Validation. In *Third International Conference onReliability, Safety, and Security of Railway Systems (RSSRail'2019)*, volume 11495 of *LNCS*, pages 23–40. Springer.

Idani, A., Ledru, Y., Ait-Wakrime, A., Ben-Ayed, R., and Collart-Dutilleul, S. (2019b). Incremental Development of a Safety Critical System Combining formal Methods and DSMLs – Application to a Railway System. In *24th International Conference on Formal Methods for Industrial Critical Systems (FMICS'2019)*, volume 11687 of *LNCS*, pages 93–109. Springer.

Idani, A., Ledru, Y., and Vega, G. (2020). Alliance of Model Driven Engineering with a Proof-based Formal Approach. *International Journal on Innovations in Systems and Software Engineering (ISSE)*, 16(3):289–307.

Idani, A., Vega, G., and Leuschel, M. (2019c). Applying Formal Reasoning to Model Transformation: The Meeduse solution. In *Proceedings of the 12th Transformation Tool Contest, co-located with STAF'2019, Software Technologies: Applications and Foundations*, volume 2550 of *CEUR Workshop Proceedings*, pages 33–44.

Jackson, D. (2002). Alloy: A Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290.

James, P., Knapp, A., Mossakowski, T., and Roggenbach, M. (2013). Designing DSLs - A Craftsman's Approach for the Railway Domain Using CASL. In *Recent Trends in Algebraic Development Techniques*. Springer.

Jensen, K. (1981). Coloured Petri Nets and the invariant-method. *Theoretical Computer Science*, 14:317–336.

Jézéquel, J.-M., Combemale, B., Barais, O., Monperrus, M., and Fouquet, F. (2013). Mashup of Meta-Languages and its Implementation in the Kermeta Language Workbench. *Software and Systems Modeling*.

Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., and Valduriez, P. (2006). Atl: A qvt-like transformation language. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 719–720, New York, NY, USA. ACM.

Jürjens, J. (2010). *Secure Systems Development with UML*. Springer-Verlag.

Jürjens, J., Lehrhuber, M., and Wimmel, G. (2005). Model-based design and analysis of permission-based security. In *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pages 224–233.

Koleini, M. and Ryan, M. (2011). A knowledge-based verification method for dynamic access control policies. In *Formal Methods and Software Engineering - 13th International Conference on Formal Engineering Methods, ICFEM 2011, Durham, UK, October 26-28, 2011. Proceedings*, volume 6991 of *LNCS*, pages 243–258. Springer.

Kont, M., Pihelgas, M., Wojtkowiak, J., Trinberg, L., and Osula, A.-M. (2018). *Insider Threat Detection Study*. The NATO Cooperative Cyber Defence Centre of Excellence.

Körner, P., Bendisposto, J., Dunkelau, J., Krings, S., and Leuschel, M. (2019). Embedding high-level formal specifications into applications. In *International Conference on Formal Methods (FM)*, volume 11800 of *LNCS*. Springer.

Körner, P., Bendisposto, J., Dunkelau, J., Krings, S., and Leuschel, M. (2020). Integrating formal specifications into applications: the prob java api. *Formal Methods in System Design*.

Kosar, T., Bohra, S., and Mernik, M. (2016). Domain-specific languages: A systematic mapping study. *Information and Software Technology*, 71:77–91.

Kuhlmann, M., Sohr, K., and Gogolla, M. (2013). Employing UML and OCL for designing and analysing role-based access control. *Mathematical Structures in Computer Science*, 23(4).

Laleau, R. and Mammar, A. (2000). An Overview of a Method and Its Support Tool for Generating B Specifications from UML Notations. In *15th IEEE International Conference on Automated Software Engineering*, pages 269–272. IEEE Computer Society Press.

Laleau, R. and Polack, F. (2002). Coming and going from uml to b: A proposal to support traceability in rigorous is development. In *2nd International Conference of B and Z Users*, volume 2272 of *LNCS*, pages 517–534. Springer.

Langer, P., Mayerhofer, T., and Kappel, G. (2014). Semantic model differencing utilizing behavioral semantics specifications. In *17th International Conference Model-Driven Engineering Languages and Systems - MODELS*, volume 8767 of *LNCS*, pages 116–132. Springer.

Lano, K., Clark, D., and Androutsopoulos, K. (2004). UML to B: Formal Verification of Object-Oriented Models. In *Integrated Formal Methods*, volume 2999 of *LNCS*. Springer.

Ledang, H. (2001). Automatic translation from uml specifications to b. In *Automated Software Engineering*, page 436.

Ledru, Y. (2006). Using jaza to animate roz specifications of UML class diagrams. In *30th Annual IEEE / NASA Software Engineering Workshop (SEW-30*, pages 253–262. IEEE Computer Society.

Ledru, Y., Idani, A., Milhau, J., Qamar, N., Laleau, R., Richier, J.-L., and Labiadh, M.-A. (2014). Validation of IS security policies featuring authorisation constraints. *International Journal of Information System Modeling and Design (IJISMD)*.

Ledru, Y., Idani, A., and Richier, J. (2015a). Validation of a security policy by the test of its formal B specification - A case study. In *3rd IEEE/ACM FME Workshop on Formal Methods in Software Engineering, FormaliSE*, pages 6–12. IEEE Computer Society.

Ledru, Y., Idani, A., Ayed, R. B., Wakrime, A. A., and Bon, P. (2019). A separation of concerns approach for the verified modelling of railway signalling rules. In Dutilleul, S. C., Lecomte, T., and Romanovsky, A. B., editors, *Third International Conference on Reliability, Safety, and Security of Railway Systems - Modelling, Analysis, Verification, and Certification (RSSRail)*, volume 11495 of *LNCS*, pages 173–190. Springer.

Ledru, Y., Idani, A., Milhau, J., Qamar, N., Laleau, R., Richier, J., and Labiadh, M. (2015b). Validation of IS security policies featuring authorisation constraints. *International Journal of Information System Modeling and Design*, 6(1):24–46.

Leroy, X. (2009). Formal verification of a realistic compiler. *Communications of the ACM*, pages 107–115.

Leuschel, M. and Butler, M. (2003). ProB: A Model Checker for B. In *FME 2003: Formal Methods Europe*, volume 2805 of *LNCS*. Springer-Verlag.

Leuschel, M. and Butler, M. (2008). Prob: an automated analysis toolset for the b method. *International Journal on Software Tools for Technology Transfer*, 10(2):185–203.

Lienhard, A., Gîrba, T., and Nierstrasz, O. (2008). Practical object-oriented back-in-time debugging. In Vitek, J., editor, *ECOOP 2008 – Object-Oriented Programming*, pages 592–615. Springer.

Mammar, A. and Frappier, M. (2015). Proof-based verification approaches for dynamic properties: application to the information system domain. *Formal Asp. Comput.*, 27(2):335–374.

Mandell, R. and Estrin, G. (1966). A meta-compiler as a design automation tool. In *Proceedings of the SHARE Design Automation Project*, DAC'66, pages 1–13, USA. ACM.

Martin, S. (2003). The Best of Both Worlds Integrating UML with Z for Software Specifications. *Journal of Computing and Control Engineering*, 14:8–11.

Mashkoor, A., Kossak, F., and Egyed, A. (2018). Evaluating the suitability of state-based formal methods for industrial deployment. *Softw. Pract. Exp.*, 48(12):2350–2379.

Matulevicius, R. and Dumas, M. (2010). Towards model transformation between secureuml and umlsec for role-based access control. In Barzdins, J. and Kirikova, M., editors, *Databases and Information Systems VI (DB&IS'10), July 5-7*, volume 224 of *Frontiers in Artificial Intelligence and Applications*, pages 339–352. IOS Press.

Mayerhofer, T., Langer, P., Wimmer, M., and Kappel, G. (2013). Towards xmof: Executable dsmls based on fuml. In *International Conference on Software Language Engineering - SLE*, volume 8225 of *LNCS*, pages 56–75. Springer.

Meeduse (2020). http://vasco.imag.fr/tools/meeduse/. Accessed: 15-12-2020.

Merilinna, J. and Pärssinen, J. (2010). Verification and validation in the context of domain-specific modelling. In *Proceedings of the 10th Workshop on Domain-Specific Modeling*, pages 9:1–9:6, New York, NY, USA. ACM.

Merkle, D. and Middendorf, M. (2006). Ant colony optimization. *European Journal of Operational Research*, 168(1):269–271.

Meyer, E. (2001). *Développements formels par objets : utilisation conjointe de B et d'UML*. PhD thesis, Université de Nancy 2.

Muller, P.-A., Fleurey, F., and Jézéquel, J.-M. (2005). Weaving executability into object-oriented meta-languages. In *Proceedings of MODELS/UML'2005*, Montego Bay, Jamaica.

Object Management Group (2011). *Semantics of a Foundational Subset for Executable UML Models (fUML)*. https://www.omg.org/spec/FUML/.

Object Management Group (2015). *Meta Object Facility (MOF) 2.5.1 Core Specification*. https://www.omg.org/spec/MOF/2.5.1/.

OMG (2003). MDA Guide Version 1.0.1. http://www.omg.org/mda/. http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf.

Oxford (2020). *The Oxford Dictionary*. Oxford University Press.

Petri, C. A. and Reisig, W. (2008). Petri net. *Scholarpedia*, 3(4):6477.

Petri-net ecore file (2020). https://github.com/gemoc/petrinet/blob/master/petrinetv1/fr.inria.diverse.sample.petrinetv1.model/model/petrinetv1.ecore. Accessed: 15-12-2020.

PNML Homepage (2020). http://www.pnml.org. Accessed: 15-12-2020.

Pouzancre, G. (2003). How to Diagnose a Modern Car with a Formal B Model? In Bert, D., Bowen, J. P., King, S., and Waldén, M., editors, *ZB 2003: Formal Specification and Development in Z and B*, pages 98–100, Berlin, Heidelberg. Springer Berlin Heidelberg.

Power, D., Slaymaker, M., and Simpson, A. (2010). On the modelling and analysis of Amazon Web Services access policies. In *Proceedings of Abstract State Machines, Alloy, B and Z (ABZ 2010)*, page 394. Springer-Verlag LNCS, volume 5977.

Probst, C. W., Hunker, J., Gollmann, D., and Bishop, M., editors (2010). *Insider Threats in Cyber Security*, volume 49 of *Advances in Information Security*. Springer.

Qamar, N., Ledru, Y., and Idani, A. (2011a). Evaluating RBAC supported techniques and their validation and verification. In *Sixth International Conference on Availability, Reliability and Security (ARES), Vienna, Austria, August 22-26*, pages 734–739. IEEE Computer Society.

Qamar, N., Ledru, Y., and Idani, A. (2011b). Validation of security-design models using Z. In Qin, S. and Qiu, Z., editors, *13th International Conference on Formal Engineering Methods (ICFEM), Durham, UK, October 26-28*, volume 6991 of *LNCS*, pages 259–274. Springer.

Radhouani, A., Idani, A., Ledru, Y., and Rajeb, N. B. (2015). Symbolic Search of Insider Attack Scenarios from a Formal Information System Modeling. *LNCS Transactions on Petri Nets and Other Models of Concurrency*, 10:131–152.

RailML (2018). RailML Latest Release 3.1. https://www.railml.org/en/home.html. Accessed: 2021-12-03.

Rivera, J., Durán, F., and Vallecillo, A. (2009). Formal specification and analysis of domain specific models using maude. *Simulation*, 85:778–792.

Sarna-Starosta, B. and Stoller, S. D. (2004). Policy analysis for security-enhanced linux. In *Proceedings of the 2004 Workshop on Issues in the Theory of Security (WITS)*, pages 1–12. Available at http://www.cs.sunysb.edu/~stoller/WITS2004.html.

Schaad, A. and Moffett, J. D. (2002). A lightweight approach to specification and analysis of role-based access control extensions. In *Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies*, pages 13–22, New York, NY, USA. Association for Computing Machinery.

Schon, W., Larraufie, G., MoÃ« ns, G., and PorÃ©, J. (2014). *Railway signalling and automation Volume 3*. La vie du rail.

Snook, C. and Butler, M. (2004). U2B-A tool for translating UML-B models into B. In Mermet, J., editor, *UML-B Specification for Proven Embedded Systems Design*.

Snook, C. and Butler, M. (2006a). UML-B: Formal Modeling and Design Aided by UML. *ACM Transactions of Software Engineering Methodologies*, 15(1):92–122.

Snook, C. and Butler, M. (2006b). UML-B: Formal modeling and design aided by UML. *ACM Transactions on Software Engineering and Methodology*, 15(1).

Sohr, K., Drouineaud, M., Ahn, G.-J., and Gogolla, M. (2008). Analyzing and managing role-based access control policies. *IEEE Transactions on Knowledge and Data Engineering*, 20(7):924–939.

Spivey, J. M. (1992). The Z Notation: A reference manual (2nd ed.). Prentice Hall.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2008). *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2nd edition.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition.

Svendsen, A., Haugen, Ø., and Møller-Pedersen, B. (2012). Synthesizing software models: Generating train station models automatically. In *Integrating System and Software Modeling*, pages 38–53. Springer.

The Standish Group (2008). Chaos report. http://www.cs.nmt.edu/~cs328/reading/Standish.pdf – last visited 15^{th} of June, 2008.

Thong, W. J. and Ameedeen, M. A. (2015). A survey of petri net tools. In *Advanced Computer and Communication Engineering Technology*, pages 537–551, Cham. Springer.

Tikhonova, U. (2017a). *Engineering the dynamic semantics of domain specific languages*. PhD thesis, Department of Mathematics and Computer Science. Proefschrift.

Tikhonova, U. (2017b). Reusable specification templates for defining dynamic semantics of dsls. *Software & Systems Modeling*.

Tikhonova, U., Manders, M., Brand, van den, M., Andova, S., and Verhoeff, T. (2013). Applying model transformation and event-b for specifying an industrial dsl. In Boulanger, F., Famelis, M., and Ratiu, D., editors, *MoDeVVa 2013 : Workshop on Model Driven Engineering, Verification and Validation : Proceedings of the 10th International Workshop on Model Driven Engineering, Verification and Validation co-located with 16th International Conference on Model Driven Engineering Miami, Florida, October 1st, 2013*, CEUR Workshop Proceedings, pages 41–50. CEUR-WS.org.

Toahchoodee, M., Ray, I., Anastasakis, K., Georg, G., and Bordbar, B. (2009a). Ensuring spatio-temporal access control for real-world applications. In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies*, pages 13–22, New York, NY, USA. Association for Computing Machinery.

Toahchoodee, M., Ray, I., Anastasakis, K., Georg, G., and Bordbar, B. (2009b). Ensuring spatio-temporal access control for real-world applications. In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies*, SACMAT '09, pages 13–22, New York, NY, USA. ACM.

Vallecillo, A. and Gogolla, M. (2017). Adding random operations to OCL. In *Proceedings of MODELS 2017 Satellite Event*, CEUR Workshop Proceedings, pages 324–328. CEUR-WS.org.

Vergu, V., Neron, P., and Visser, E. (2015). DynSem: A DSL for Dynamic Semantics Specification. In Fernández, M., editor, *26th International Conference on Rewriting Techniques and Applications (RTA 2015)*, volume 36 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 365–378. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Voelter, M., Kolb, B., Birken, K., Tomassetti, F., Alff, P., Wiart, L., Wortmann, A., and Nordmann, A. (2019). Using language workbenches and domain-specific languages for safety-critical software development. *Softw. Syst. Model.*, 18(4):2507–2530.

Vu, L., Haxthausen, A., and Peleska, J. (2014). A domain-specific language for railway interlocking systems. *10th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems*, pages 200–209.

Wachsmuth, G. (2008). Modelling the operational semantics of domain-specific modelling languages. In Lämmel, R., Visser, J., and Saraiva, J., editors, *Generative and Transformational Techniques in Software Engineering II (GTTSE)*, pages 506–520. Springer Berlin Heidelberg.

Wagelaar, D. and Straeten, R. V. D. (2006). A comparison of configuration techniques for model transformations. In *ECMDA-FA*, pages 331–345.

Wakrime, A. A., Ayed, R. B., Dutilleul, S. C., Ledru, Y., and Idani, A. (2018). Formalizing railway signaling system ERTMS/ETCS using uml/event-b. In Abdelwahed, E. H., Bella-

treche, L., Golfarelli, M., Méry, D., and Ordonez, C., editors, *8th International Conference on Model and Data Engineering - MEDI*, volume 11163 of *LNCS*, pages 321–330. Springer.

Warmer, J. and Kleppe, A. (1999). *The Object Constraint Language: Precise Modeling with UML*. Object Technology Series. Addison-Wesley.

Wildmoser, M. and Nipkow, T. (2004). Certifying machine code safety: Shallow versus deep embedding. In Slind, K., Bunker, A., and Gopalakrishnan, G., editors, *Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223 of *LNCS*, pages 305–320. Springer.

Yangui, R. (2016). *Modélisation UML/B pour la validation des exigences de sécurité des règles d'exploitation ferroviaires*. PhD thesis.

Yu, L., France, R., Ray, I., and Ghosh, S. (2009). A Rigorous Approach to Uncovering Security Policy Violations in UML Designs. In *2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, pages 126–135.

Yuan, C., He, Y., He, J., and Zhou, Z. (2006). A verifiable formal specification for RBAC model with constraints of separation of duty. In *Inscrypt*, pages 196–210. LCNS 4318, Springer.

Zalila, F., Crégut, X., and Pantel, M. (2013). Formal verification integration approach for dsml. In *Model-Driven Engineering Languages and Systems*, pages 336–351. Springer.

Zao, J., Wee, H., Chu, J., and Jackson, D. (2003). RBAC schema verification using lightweight formal model and constraint analysis. In *Proceedings of the 8th ACM Symposium on Access Control Models and Technologies*. ACM.

Zhang, N., Ryan, M., and Guelev, D. P. (2005). Evaluating access control policies through model checking. In *Information Security, 8th International Conference, ISC 2005, Singapore, September 20-23, 2005, Proceedings*, volume 3650 of *LNCS*, pages 446–460.

Zhang, N., Ryan, M., and Guelev, D. P. (2008). Synthesising verified access control systems through model checking. *Journal of Computer Security*, 16(1):1–61.

Abstract

My research works are dedicated to the integration of two well known paradigms: Formal Methods (FM) and Model-Driven Engineering (MDE). This integration is called Formal MDE (FMDE) all along the current document. In fact, several works have been already done in order to strengthen the MDE paradigm with formal reasoning, and therefore make it more viable as far as safety and security concerns have to be addressed. When taken separately, these works provide a partial coverage of MDE, but when combined they can address a wide range of models and languages. During the last decade, I investigated two directions in which the FMDE paradigm proved its value: (*i*) Model-Driven Security (MDS), and (*ii*) Domain-Specific Languages (DSLs). Under the MDE umbrella, both the MDS and DSL communities advocate for the use of models throughout the development process, providing solutions to the validation problem ('*do the right system*'). Nonetheless, the verification problem ('*do the system right*') is still a major challenge, perhaps because formal reasoning (*i.e.* model-checking and/or theorem proving) was not apart of the MDE initiative. To be pragmatic my contributions build on well-established notations: mainly UML and B, and − at a smaller scale − BPMN, CSP, Z and Petri-Nets. Besides, the obtained results can be inspiring and, in my opinion, should be extended with other (semi-)formal languages, which would confer to FMDE a broader spectrum. This document summarizes for every research direction (respectively MDS and DSLs) the challenges that guided my works, and give an overview of my contributions and publications in the field.