A testing framework for executable domain-specific languages

Faezeh Khorram

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Thèse de doctorat de

L'École Nationale Supérieure Mines-Télécom Atlantique
Bretagne Pays de la Loire - IMT Atlantique

École Doctorale N°601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique
Par

Faezeh KHORRAM

A Testing Framework for Executable Domain-Specific Languages

Thèse présentée et soutenue à IMT Atlantique - Campus de Nantes, le 12 Decembre 2022
Unité de recherche : Laboratoire des Sciences du Numérique de Nantes (LS2N)
Thèse N°2022IMTA0332

Rapporteurs avant soutenance :

M^me^ Anne ETIEN	Professeur des universités, Université de Lille, France
M. Juergen DINGEL	Professor, Queen's University, Canada

Composition du Jury

Président :	M. Antoine BEUGNARD	Professeur, IMT Atlantique, France
Examinateurs :	M^me^ Anne ETIEN	Professeur des universités, Université de Lille, France
	M. Juergen DINGEL	Professor, Queen's University, Canada
	M. Benoit BAUDRY	Professor, KTH Royal Institute of Technology, Sweden
	M. Javier TROYA	Associate Professor, Universidad de Málaga, Spain
	M. Erwan BOUSSE	Maître de conférences, Nantes Université, France
Directeur de thèse :	M. Gerson SUNYE	Maître de conférences HDR, Nantes Université, France
Co-encadrant de thèse :	M. Jean-Marie MOTTU	Maître de conférences, Nantes Université, France

Acknowledgments

This thesis reached its final milestone thanks to the support of many people. I should start with my parents who did whatever they could to build the path for their children's growth, my family members who encouraged me along the way, and my husband who gave me the strength and courage to tackle the challenges. My sincere thanks to them.

I heartily appreciate Dr. Ali Pouyan and Dr.Raman Ramsin for recognizing my potential a long time ago, educating me on how to think wider and deeper, and motivating me to have higher goals for my life.

I would use this opportunity to express my special gratitude to my dear supervisors Prof. Gerson Sunyé, Dr. Jean-Marie Mottu, and Dr. Erwan Bousse for their constant support, guidance, and patience. I am also thankful to all NaoMod members especially Dr. Massimo Tisi for the positive energy and motivation that he speared into the team.

I was fortunate to be a Marie-Curie Early Stage Researcher in the Lowcomote European project which gave me many wonderful international opportunities. Thanks to the Lowcomote, I had international collaborations one with JKU university of Linz under the supervision of Prof. Manuel Wimmer, and one with the Universidad Autónoma de Madrid under the supervision of Prof. Juan de Lara, and Prof. Esther Guerra. My special thanks to them and to Dr. Antonio Garmendia and Dr. Pablo Gómez-Abajo who had an undeniable role in the success of our collaborations.

Finally, many thanks to the jury members, Prof. Anne Etien, Prof. Juergen Dingel, Prof. Benoît Baudry, Prof. Antoine Beugnard, and Dr. Javier Troya for their thoughtful questions which brought up many new ideas and for awarding me the precious title of Dr.

Dedication

To Yousef, you were besides me all along the way.
To my parents, we were waiting for this moment for a long time.

Résumé long en français

Contexte

Les logiciels occupent le rôle central de l'ère de la transformation numérique. Ils sont utilisés de manière exponentielle dans des domaines complexes tels que les systèmes cyber-physiques (CPS) ou l'Internet des objets (IoT). Cependant, la croissance de la complexité entrave la collaboration productive entre les personnes impliquées dans le cycle de vie du développement logiciel. En particulier, en raison de l'émergence de domaines interdisciplinaires, les parties prenantes apportent généralement des compétences diverses provenant de plusieurs domaines. Parvenir à une compréhension commune du système par tous les acteurs est alors difficile.

La modélisation des logiciels et des systèmes est aujourd'hui considérée comme une solution viable pour relever ces défis. Elle permet d'avoir différentes représentations d'un même système pour différents objectifs grâce à des moyens d'abstraction adaptés. L'ingénierie dirigée par les modèles (IDM) est un paradigme de développement logiciel qui utilise les modèles comme artefacts de développement essentiels [125]. En particulier, l'IDM encourage l'utilisation de modèles au moment de la conception et propose ensuite des techniques de transformation de modèles et de génération de code pour automatiser leur transition vers un système déployable [35]. Les processus IDM reposent généralement sur des langages de modélisation dédiés (LMD) pour la définition des modèles [88, 104]. Les LMDs sont des langages logiciels conçus pour des domaines techniques ou applicatifs spécifiques, et donc adaptés pour être utilisés par des experts du domaine [56]. Bien que ces langages permettent notamment aux experts du domaine de s'impliquer dans le cycle de vie du développement, un défi est que pour chaque LMD nouvellement développé, un environnement de modélisation complet doit être fourni pour utiliser efficacement le LMD [51, 101]. Ainsi, des plateformes dédiées appelées ateliers de langage ont émergé pour définir des langages qui peuvent directement bénéficier d'un support d'outils prêts à l'emploi [29, 51]. Le succès de l'IDM dans la gestion de la complexité et dans l'implication directe des experts du domaine dans le cycle de vie du développement a conduit à son adoption dans différents domaines industriels [73, 151] ainsi qu'à l'émergence de plateformes de développement Low-Code

(LCDP) [48, 138, 144]. Un LCDP est un environnement de développement basé sur l'IDM, généralement sur le cloud, qui peut être utilisé par des non-programmeurs pour créer des applications logicielles. Comme les LCDP ciblent généralement différents domaines d'application, ils offrent différents LMD pour développer des applications.

L'IDM réduit la quantité de travail manuel par le biais d'une automatisation de nombreuses tâches, ce qui permet de réduire les erreurs humaines dans le système final. Cependant, les modèles de conception peuvent toujours comporter des défauts et si ces défauts ne sont pas résolus au début de la phase de conception, ils seront propagés à tous les modèles/codes ultérieurement générés à partir des modèles de conception. S'il est parfois possible de remonter depuis un défaut découvert dans le système final jusqu'aux modèles sources, il s'agit d'un effort très complexe, long et coûteux. Il est donc nécessaire de procéder à la vérification et à la validation (V&V) des modèles de conception. Parmi les techniques de V&V existantes, certaines sont utilisées pour évaluer les caractéristiques statiques des modèles, comme la conformité des noms d'entités aux normes de nommage. Cependant, pour les modèles de conception décrivant les aspects dynamiques des systèmes, appelés modèles *comportementaux*, des techniques de V&V dynamiques sont nécessaires. Ces techniques reposent sur l'exécution des modèles, ce qui signifie que leur application réservée aux LMDs qui comportent une sémantique d'exécution, tels que les LMDs avec une sémantique translationnelle (c'est-à-dire la compilation vers un langage exécutable) ou une sémantique opérationnelle (c'est-à-dire l'interprétation directe). Dans cette thèse, nous nous concentrons sur les LMDs à sémantique opérationnelle, appelés *LMDs exécutables (LMDx)*.

Énoncé du problème

Parmi les techniques de V&V dynamique, le test est la principale méthode utilisée pour évaluer les systèmes logiciels [12]. Le test consiste à exécuter des systèmes dans des scénarios intéressants et d'observer s'ils se comportent comme prévu. Un LMDx doté d'outils de test permet à ses utilisateurs, c'est-à-dire les experts du domaine, d'examiner l'exactitude des comportements modélisés le plus tôt possible. Cependant, parmi les nombreux LMDx existants, seuls quelques-uns offrent des outils de test [74, 75, 87, 97]. Étant donné un LMDx, ses experts de domaine peuvent tester les modèles conformes si (i) les *concepts de domaine* peuvent être utilisés dans la spécification des cas de test ; et (ii) les cas de test peuvent être exécutés à l'unisson avec les modèles testés. Les concepts de domaine et les moyens d'exécution diffèrent généralement d'un LMDx à l'autre. Cette diversité et cette hétérogénéité entraînent un travail coûteux, sujet aux erreurs et non réutilisable lors de la fourniture d'un support de test pour un LMDx [101]. Par conséquent,

dans un contexte où l'ingénierie de nouveaux LMDx est récurrente, il est nécessaire d'adopter une approche *systématique* pour fournir un support de test pour chaque LMDx donné.

Une solution prometteuse consiste à disposer d'un environnement de test qui soit *générique* en ce qui concerne les LMDx qu'il supporte, et qui soit en même temps utilisable par les experts du domaine de tout LMDx donné. Le principal défi à relever pour proposer un tel environnement de test est de fournir un langage de test répondant à trois exigences :

- **Req#1** : il doit permettre aux experts du domaine d'écrire des cas de test pour les modèles en permettant l'utilisation des concepts du domaine pour définir (i) comment un modèle sous test doit être exécuté ; et (ii) quels résultats doivent être attendus de l'exécution.

- **Req#2** : il doit être capable de lancer l'exécution du modèle à tester selon les besoins des cas de test.

- **Req#3** : il doit fournir des moyens de vérifier si le modèle testé se comporte comme prévu par les cas de test.

En résumé, un environnement de test générique répond à ces exigences s'il peut adapter automatiquement son langage de test aux concepts, aux utilisateurs et aux possibilités d'exécution d'un LMDx donné.

Le fait de pouvoir écrire et exécuter des cas de test pour des modèles ne garantit pas l'efficacité des cas de test définis. Dans le domaine des langages de programmation, plusieurs activités de test avancées sont généralement réalisées pour améliorer l'efficacité des tests, telles que (i) la mesure de la qualité des tests pour s'assurer que les cas de test définis sont suffisamment bons [12], (ii) le diagnostic des tests échoués pour localiser les fautes [150], (iii) ou encore l'amélioration des tests pour renforcer les cas de test à différentes fins, comme la détection des défauts de régression [42]. Les techniques existantes pour mener à bien ces activités sont principalement développées pour chaque langage individuellement et sont fondées sur leurs environnements de test respectifs, car elles doivent manipuler directement les cas de test et le système testé. Par conséquent, l'exploitation de techniques de test avancées dans le contexte des LMDx se heurte à nouveau au défi de la diversité et de l'hétérogénéité des LMDx. Néanmoins, un avantage supplémentaire de la proposition d'un environnement de test générique pour les LMDx, tel que décrit précédemment, est de permettre l'adaptation de ces techniques supplémentaires directement pour tout LMDx. Nous considérons donc trois exigences supplémentaires pour un environnement de test convaincant :

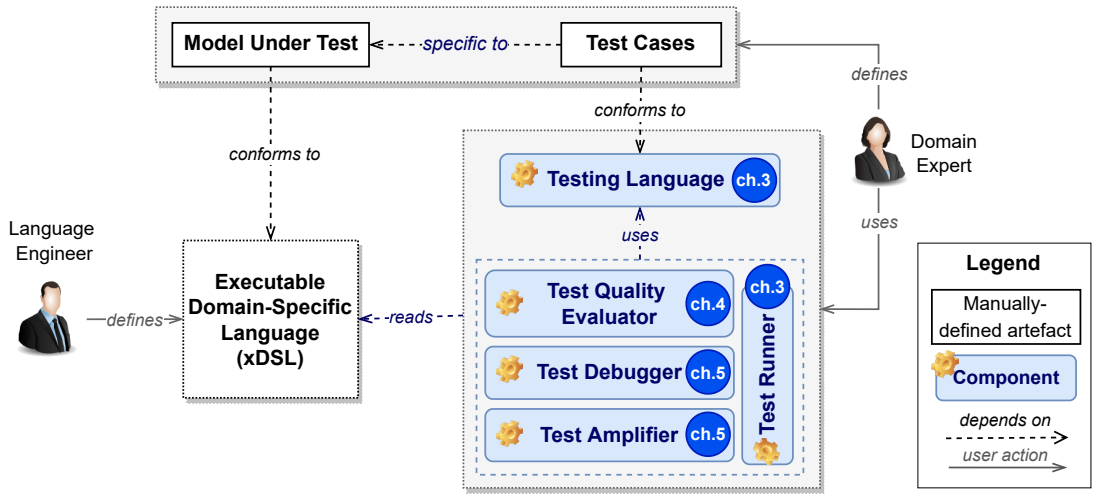


FIGURE 1 : Une vue d'ensemble de l'environnement proposé mettant en œuvre les contributions de la thèse

- **Req#4** : il doit fournir des moyens de mesure de la qualité des tests pour évaluer la capacité des cas de test du modèle à (i) s'appliquer à différentes parties d'un modèle ; et (ii) détecter les défauts potentiels du modèle.
- **Req#5** : il doit permettre aux experts du domaine de déboguer un cas de test qui a échoué afin de localiser le défaut du modèle testé.
- **Req#6** : il doit améliorer automatiquement les cas de test écrits par les experts du domaine pour renforcer leur capacité de détection des erreurs.

Contributions

Dans cette thèse, nous proposons plusieurs contributions pour répondre aux exigences susmentionnées, toutes mises en œuvre dans **un nouvel environnement de test générique pour les LMDx**. La figure 1 présente une vue d'ensemble de l'environnement que nous proposons. Nous considérons un *ingénieur de language* qui définit un LMDx (à gauche) et a besoin d'un environnement de test pour son LMDx afin de permettre aux *experts de domaine*, qui utilisent le LMDx pour définir des modèles (à droite), de tester efficacement leurs modèles.

Pour permettre la définition de cas de test pour les modèles exécutables (Req#1), l'environnement s'appuie sur le Test Description Language (TDL), un langage standardisé pour la spécification des descriptions de test [98]. Ce choix a été fait pour deux raisons principales. Premièrement, TDL n'est pas lié à un langage particulier pour la définition du système à tester, il représente donc un candidat

intéressant pour l'écriture générique de cas de test pour tout modèle exécutable. Deuxièmement, TDL a été conçu comme un langage simple pour les testeurs n'ayant pas de connaissances en programmation, ce qui en fait un bon choix pour les experts du domaine travaillant sur des modèles. Cependant, pour appliquer TDL à un LMDx donné, l'expert du domaine doit d'abord définir les concepts nécessaires spécifiques au domaine, avant de pouvoir écrire des cas de test pour les modèles définis par le LMDx. Pourtant, cette tâche exige une connaissance des éléments internes du LMDx et, comme il est peu probable que les experts du domaine les connaissent, il s'agit d'une tâche coûteuse et sujette aux erreurs. Dans l'environnement que nous proposons, nous générons automatiquement une bibliothèque TDL spécifique au domaine à partir de la définition d'une LMDx donnée. Une telle bibliothèque peut être utilisée par l'expert du domaine pour écrire des cas de test pour les modèles conformes au LMDx considéré (répondant à la Req#1). Nos contributions pour la définition des cas de test donnent lieu au composant **Testing Language** de l'environnement (en haut au centre de la figure 1) et seront détaillées dans le chapitre 3.

Une fois les cas de test définis, ils doivent être exécutés sur les modèles testés (Req#2) et les résultats doivent être produits en vérifiant si les modèles se sont comportés comme prévu par les cas de test (Req#3). L'exécution de ces tâches complexes nécessite des connaissances sur la façon d'exécuter un modèle et d'observer son comportement, ce qui est en fait fourni par le LMDx auquel le modèle testé se conforme. Comme TDL est conçu comme un standard indépendant du langage, il ne fournit aucun support pour ces tâches. Dans cette thèse, nous proposons une sémantique opérationnelle pour TDL qui est capable d'exécuter des cas de test sur des modèles exécutables tout en étant découplée de tout LMDx spécifique (répondant à la Req#2). Elle fournit également plusieurs facilités pour interroger le comportement d'un modèle lors de son exécution par un cas de test et ainsi produire les résultats de l'exécution du test (répondant à la Req#3). Le composant **Test Runner** de l'environnement (dans le coin inférieur droit de la Figure 1) offre les facilités d'exécution des tests qui seront présentées dans le Chapitre 3.

Afin de mesurer correctement la qualité des cas de test écrits (Req#4), l'environnement de test que nous proposons fournit pour les LMDx à la fois des moyens pour effectuer un *calcul de la couverture* [12] ainsi qu'une *analyse de mutation* [79], deux techniques déjà bien connues dans le domaine des langages de programmation. La première mesure la part du modèle à tester qui est exercée par un cas de test TDL donné, tandis que la seconde analyse la capacité des cas de test à trouver les défauts potentiels du modèle (répondant à la Req#4). Pour les deux approches, les résultats d'exécution des tests produits par le composant *Test Runner* sont analysés pour calculer la qualité des cas de test exécutés. Le composant **Test Quality Evaluator** de l'environnement (au centre de la figure 1) réalise cette

partie de nos contributions, présentée plus loin dans le chapitre 4.

Si les cas de test échouent sur un modèle, en supposant que le cas de test est correct, il y a un défaut dans le modèle testé qui doit être corrigé. L'environnement proposé aide les experts du domaine à localiser les défauts des modèles testés (Req#5) en proposant deux approches. Tout d'abord, il offre une approche manuelle basée sur le débogage interactif permettant aux experts du domaine de déboguer un scénario de test en même temps que son modèle testé afin d'observer progressivement la réaction du modèle à la réception des requêtes du scénario de test. Deuxièmement, comme le débogage manuel est fastidieux pour les cas de test de modèles complexes et/ou de grande taille, l'environnement proposé fournit également une adaptation de l'approche de localisation de défauts appelée "Spectrum-based Fault Localization" (SBFL) [150]. SBFL est une approche automatique qui calcule la probabilité que chaque partie d'un programme (e. g., une instruction d'un programme Java) soit défectueuse, sur la base des résultats des cas de test et de leurs informations de couverture correspondantes. Les contributions mentionnées précédemment pour tester des modèles et mesurer leur couverture nous ont permis d'adapter SBFL au contexte des LMDx. L'approche proposée calcule le classement des éléments des modèles en fonction du degré de suspicion à partir de deux ingrédients : les résultats des tests produits par le *Test Runner* et les mesures de couverture générées par le *Test Quality Evaluator* (répondant à la Req#5). Le composant **Test Debugger** de l'environnement (au centre de la Figure 1) fournit les deux approches de débogage proposées qui seront présentées dans le Chapitre 5.

Le test et le débogage d'un modèle contribuent à le rendre correct dans une version donnée, mais il existe toujours un risque de régression, c'est-à-dire des fautes ajoutées lors des futures évolutions du modèle. Les experts du domaine peuvent mesurer la capacité de leurs jeux de tests à détecter les fautes potentielles grâce à nos outils d'analyse de mutation. Lorsque les cas de test existants ne sont pas assez solides, il est nécessaire d'améliorer les tests pour pouvoir détecter l'apparition de régressions, ce qui est une tâche complexe pour les experts du domaine (Req#6). L'environnement proposé fournit une approche générique d'amplification des tests qui améliore automatiquement la capacité des cas de test à détecter les régressions. Plus précisément, il génère de nouveaux cas de test en modifiant les cas de test existants écrits manuellement à l'aide d'un ensemble de modificateurs proposés, puis évalue le niveau d'amélioration à l'aide d'une analyse de mutation (répondant à la Req#6). Le composant **Test Amplifier** (au centre de la figure 1) ajoute à l'environnement les fonctionnalités d'amplification de test que nous proposons et qui seront présentées au chapitre 5.

Méthodologie de recherche

Pour mener cette recherche, nous avons suivi la méthodologie de recherche de la science du design (DSRM) [118] et les lignes directrices de la science du design [70]. Nous avons en effet réalisé les six activités principales de l'approche DSRM pour chaque partie de nos contributions qui seront détaillées dans chaque chapitre de cette thèse :

1. *Identification et motivation du problème* : Nous avons étudié l'espace du problème en fournissant son contexte et en rassemblant une vue d'ensemble des approches existantes dans le contexte des environnements de test pour les LMDx. Enfin, nous avons identifié les forces et les limites de l'état de l'art.

2. *Définir les objectifs d'une nouvelle solution* : Nous avons identifié les principales exigences pour fournir une approche systématique aux ingénieurs linguistiques afin de fournir un environnement de test pour leurs LMDx.

3. *Conception et développement* : Nous avons conçu un environnement de test générique pour les LMDx qui répond aux exigences identifiées.

4. *Démonstration* : Nous avons implémenté l'environnement proposé pour le GEMOC Studio, un atelier de langage et de modélisation pour les LMDx [29].

5. *Evaluation* : Pour évaluer la généricité de l'environnement en ce qui concerne les LMDx supportés, nous avons évalué empiriquement chacun des composants proposés sur plusieurs LMDx de différents domaines d'application.

6. *Communication* : Les résultats de cette thèse sont publiés dans deux revues scientifiques, deux conférences de haut niveau et un workshop.

Contexte de la thèse

Cette thèse a été réalisée dans le cadre du projet européen Lowcomote[1] [138], un réseau de formation innovant (ITN) pour former la prochaine génération d'experts en plateformes d'ingénierie low-code évolutives. Une partie des contributions est le résultat de détachements internationaux prévus par le projet Lowcomote. Premièrement, le calcul de la couverture et la localisation automatique des défauts (i. e., SBFL) ont été réalisés en collaboration avec le département d'informatique de gestion - génie logiciel (WIN-SE) de l'université JKU (Linz, Autriche). Deuxièmement,

[1] www.lowcomote.eu, This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement n°813884.

l'analyse des mutations et l'amplification automatique des tests sont le résultat d'une collaboration avec le groupe de recherche Miso de l'Université autonome de Madrid (Madrid, Espagne).

Contents

Chapter 1

Introduction

1.1 Context

Software applications are the main player in the era of digital transformation. They are exponentially used in complex domains such as Cyber-Physical Systems (CPS) or the Internet of Things (IoT). The growth of complexity hinders productive collaboration among the actors of the software development lifecycle. Especially, due to the emergence of interdisciplinary domains, stakeholders usually provide diverse expertise from various domains, hence providing a shared understanding for all of them has turned into a challenge.

Software and systems modeling has been seen as a viable solution to tackle these challenges. It allows having different representations of the same system for various objectives through convenient means of abstraction. Model-Driven Engineering (MDE) is a software development paradigm that uses models as pivotal development artifacts [125]. In particular, MDE promotes the use of models at design time and then offers model transformation and code generation techniques to automatize their transition to a deployable system [35]. MDE processes commonly rely on Domain-Specific Languages (DSLs) for the definition of models [88, 104]. DSLs are software languages made for specific technical or application domains, thus tailored to be used by domain experts [56]. While these languages provide specific support for the domain experts to involve them in the development lifecycle, one challenge is that for each newly developed DSL, a complete modeling environment has to be provided to efficiently and effectively use the DSL [51, 101]. Thus, dedicated platforms called language workbenches have emerged to define languages that can directly benefit from out-of-the-box tool support [29, 51]. The success of MDE in managing complexity and in the direct involvement of domain experts in the development lifecycle yields its adoption in different industrial domains [73, 151] as well as to the emergence of Low-Code Development Platforms (LCDPs) [48,

138, 144]. An LCDP is an MDE-based development environment typically on the cloud that can be used by non-programmers to build software applications. As LCDPs usually target different application domains, they offer different DSLs to develop applications.

MDE reduces the amount of manual work by providing automation (e. g., automatic code generation), and this ultimately results in fewer human mistakes in the final system. However, the design models may still have faults and if such faults are not resolved early in the design phase, they will be propagated to all the subsequent models/code generated from design models. While it is sometimes possible to trace a fault discovered in the final system back to the source models, it is a very complex, time-consuming, and expensive endeavor. Therefore, there is a need for performing *early* Verification & Validation (V&V) of design models. Among the existing V&V techniques, some are used to evaluate the static features of models, such as asserting that the entity names conform with the naming standards. However, the design models describing dynamic aspects of systems, the so-called *behavioral* models, require dynamic V&V techniques. These techniques rely on the execution of the models, which means their application is supported by DSLs with execution semantics, such as DSLs with translational semantics (i. e., compiling to an executable language) or operational semantics (i. e., direct interpretation). In this thesis, we focus on DSLs with operational semantics, referred to as *executable DSLs (xDSLs)*.

1.2 Problem Statement

Among dynamic V&V techniques, testing is the primary method used for evaluating software systems [12]. It involves executing systems in interesting scenarios and observing whether they act as expected. An xDSL with testing facilities allows its users, i. e., the domain experts, to investigate the correctness of their modeled behavior as early as possible. However, among many existing xDSLs, only a few provide testing facilities [74, 75, 87, 97]. Given an xDSL, its domain experts can test the conforming models if (i) the *domain concepts* can be used in the specification of test cases; and (ii) the test cases can be executed in unison with the models under test. Both the domain concepts and the execution facilities usually differ from one xDSL to another and this diversity and heterogeneity cause costly, error-prone, and non-reusable work when providing testing support for an xDSL [101]. Therefore, in a context where the engineering of new xDSLs is recurrent, there is a need for a *systematic* approach to provide testing support for every given xDSL.

One promising solution is having a testing framework that is *generic* regarding its supported xDSLs and at the same time is usable by the domain experts of any given xDSL. The main challenge when proposing such a testing framework is

providing a testing language that meets three requirements:

- **Req#1**: it must enable the domain experts to write test cases for models by allowing the use of the domain concepts in defining (i) how a model under test should be executed; and (ii) what results should be expected from the execution.

- **Req#2**: it must be able to launch the execution of the model under test as needed by the test cases.

- **Req#3**: it must provide facilities to investigate whether the model under test behaves as expected by the test cases.

In summary, a generic testing framework fulfills these requirements if it can automatically adapt its testing language to the concepts, users, and execution facilities of a given xDSL.

Being able to write and run test cases for models does not guarantee efficient testing of models. In the realm of programming languages, several advanced testing activities are usually performed to improve test efficiency, such as test quality measurement to make sure the defined test cases are good enough [12], test failure diagnosis to find the location of faults when test cases fail [150], and test improvement to strengthen the test cases for different purposes such as in detecting regression faults [42]. The existing techniques for carrying out these activities are mainly developed for each language individually and are founded on their supporting testing frameworks because they need to directly manipulate test cases and their system under test. Therefore, leveraging advanced testing techniques for the context of xDSLs faces again the challenge of xDSLs diversity and heterogeneity. Nevertheless, an additional benefit of proposing a generic testing framework for xDSLs as described earlier is enabling the adaptation of these supplementary techniques directly for any xDSL as well. So we consider three more requirements for a compelling testing framework:

- **Req#4**: it should provide test quality measurement facilities for evaluating the ability of the model's test cases in (i) exercising different parts of a model; and (ii) detecting the potential model's faults.

- **Req#5**: it should enable the domain experts to debug a failed test case in order to localize the fault of the model under test.

- **Req#6**: it should automatically improve the test cases written by the domain experts to strengthen their fault detection ability.

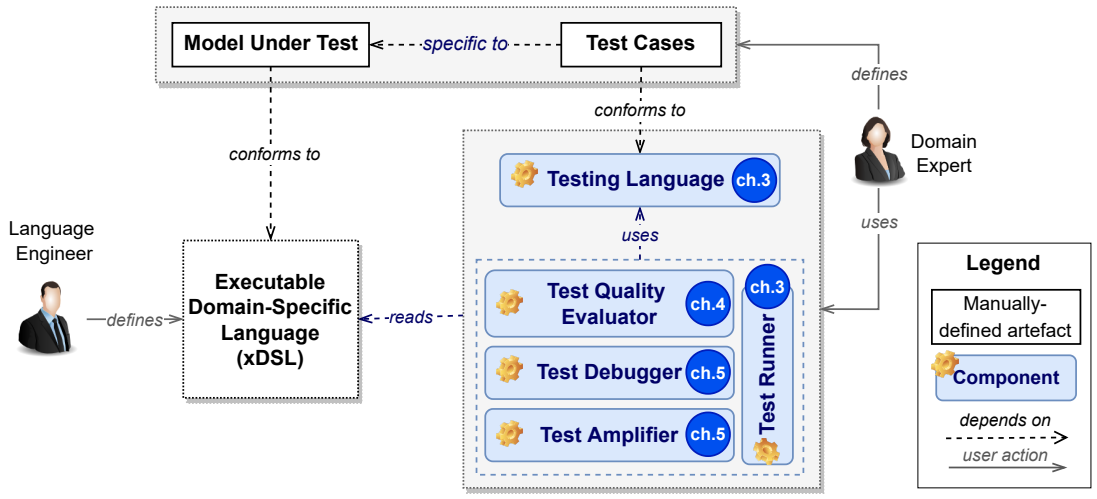


Figure 1.1: An overview of the proposed framework implementing the thesis contributions

1.3 Contributions

In this thesis, we propose several contributions to fulfill the aforementioned requirements, altogether implemented in **a novel generic testing framework for xDSLs**. Figure 1.1 presents an overview of our proposed framework. We consider a *language engineer* who defines an xDSL (on the left) and needs testing support for his/her xDSL in order to enable the *domain experts*, who use the xDSL for defining models (on the right), to efficiently test their models.

To offer facilities for the definition of test cases for executable models (Req#1), the framework relies on the Test Description Language (TDL), a standardized language for the specification of test descriptions [98]. This choice was made for two main reasons. First, TDL is not tied to any particular language for the definition of the system under test, hence it represents an interesting candidate for generically writing test cases for any executable model. Second, TDL was designed as a simple language for testers lacking programming knowledge, making it a good fit for domain experts working on models. However, to apply TDL to a given xDSL, the domain expert must first define the needed domain-specific concepts, before being able to write test cases for the models defined by the xDSL. Yet, this endeavor requires having knowledge of the internals of the xDSL and since the domain experts are unlikely to be knowledgeable of it, this is a costly and error-prone task for them. In our proposed framework, we automatically generate a domain-specific TDL library from the definition of a given xDSL. Such a library can be used by the domain expert to write test cases for models conforming to the considered xDSL (fulfilling Req#1). Our contributions for test case definition

yields to the **Testing Language** component of the framework (on top center of Figure 1.1) and will be detailed in Chapter 3.

Once test cases are defined, they must be executed against the models under test (Req#2) and the results must be produced by verifying whether the models behaved as expected by the test cases (Req#3). Performing these complex tasks requires knowledge of how to run a model and how to observe its behavior which is indeed provided by the xDSL that the model under test conforms to. As TDL is designed as a language-independent standard, it does not provide any support for them. In this thesis, we propose an operational semantics for TDL that is able to run test cases on executable models while being decoupled from any specific xDSL (fulfilling Req#2). It also provides several facilities to interrogate the behavior of a model in its execution by a test case and thus produce the test execution results (fulfilling Req#3). The **Test Runner** component of the framework (on the bottom right corner of Figure 1.1) offers the test execution facilities that will be presented in Chapter 3.

To properly measure the quality of the written test cases (Req#4), our proposed testing framework provides for xDSLs both *Coverage Computation* [12] and *Mutation Analysis* [79], two techniques already well known in the realm of programming languages. The former measures how much of the model under test is exercised by a given TDL test case, while the latter analyzes the ability of the test cases in finding potential model faults (fulfilling Req#4). For both approaches, the test execution results produced by the *Test Runner* component are analyzed to calculate the quality of the executed test cases. The **Test Quality Evaluator** component of the framework (at the center of Figure 1.1) realizes this part of our contributions, later introduced in Chapter 4.

If test cases fail on a model, assuming the test case is correct, there is a fault in the model under test that must be fixed. The proposed framework helps the domain experts in localizing the faults of the tested models (Req#5) by proposing two approaches. First, it offers a manual approach based on interactive debugging allowing the domain experts to debug a test case along with its model under test at the same time to observe gradually the model's reaction to the reception of requests from the test case. Second, as manual debugging is cumbersome for the test cases of large and/or complex models, the proposed framework provides an automatic approach using Spectrum-Based Fault Localization (SBFL) techniques [150]. SBFL is an automatic approach that calculates the probability of each program's component (e. g., statements of a Java program) being faulty, based on the results of test cases and their corresponding coverage information. Our generic facilities for testing models and measuring their coverage enabled us to adapt SBFL for the context of xDSLs. The proposed approach indeed calculates the suspiciousness-based ranking of the model elements using two ingredients: the test results produced by the *Test*

Runner and the coverage measurements generated by the *Test Quality Evaluator* (fulfilling Req#5). The **Test Debugger** component of the framework (at the center of Figure 1.1) provides both proposed debugging approaches that will be presented in Chapter 5.

Testing and debugging a model under test ensures the correctness of its current version, but there is always a threat of regression faults i. e., the faults that may occur in future updates of the model. The domain experts can measure the strength of their written test cases in detecting potential faults through our provided mutation analysis facilities. When the existing test cases are not strong enough, test improvement is required to make the model safe from regression faults which is a complex task for the domain experts (Req#6). The proposed framework provides a generic test amplification approach that automatically improves the ability of the written TDL test cases in detecting regression faults. More specifically, it generates new test cases by modifying existing manually-written test cases using a set of proposed modifiers and then evaluates the level of improvement using mutation analysis (fulfilling Req#6). The **Test Amplifier** component (at the center of Figure 1.1) adds our proposed test amplification facilities to the framework which will be presented in Chapter 5.

1.4 Research Methodology

To conduct this research, we have followed the Design Science Research Methodology (DSRM) [118] and the guidelines for design science [70]. We indeed performed the six main activities of the DSRM approach for each part of our contributions which will be detailed in each chapter of this thesis:

1. *Problem identification and motivation*: We investigated the problem space by providing its related background and gathering an overview of the existing approaches in the context of testing support for xDSLs. At the end, we have identified the strengths and the limitations of the state-of-the-art.

2. *Defining the objectives of a new solution*: We identified the main requirements for providing a systematic approach for language engineers to support their xDSLs with testing facilities.

3. *Design and development*: We designed a generic testing framework for xDSLs that fulfills the identified requirements.

4. *Demonstration*: We implemented the proposed framework for the GEMOC Studio, a language and modeling workbench for xDSLs [29].

5. *Evaluation*: To assess the genericity of the framework regarding its supported xDSLs, we empirically evaluated each of the proposed components on several xDSLs of different application domains.

6. *Communication*: The results of this thesis are published in two scientific journals, two top-rank conferences, and one workshop.

1.5 Context of the Thesis

This thesis has been carried out as a part of the Lowcomote European project[1] [138], an Innovative Training Network (ITN) for training the next generation of experts in scalable low-code engineering platforms. Parts of the contributions are the outcome of international secondments planned by the Lowcomote project. First, the coverage computation and the automatic fault localization (i. e., SBFL) parts were done in collaboration with the Department of Business Informatics – Software Engineering (WIN-SE) at JKU University (Linz, Austria). Second, the mutation analysis and the automatic test amplification parts are the results of collaboration with the Miso research group of Universidad Autónoma de Madrid (Madrid, Spain).

1.6 Outline of the Thesis

We initially provide the background and a running example along with the state-of-the-art in the context of this thesis in Chapter 2. The contributions of this thesis are then presented in chapters 3, 4, and 5:

- *Chapter 3*: presenting the provided facilities for the definition (Section 3.4) and execution (Section 3.5) of test cases for executable models.
- *Chapter 4*: introducing the test quality measurement facilities, including the proposed coverage computation approach (Section 4.2) and the mutation analysis support (Section 4.3).
- *Chapter 5*: explaining the manual (Section 5.3) and automatic (Section 5.4) test debugging facilities as well as the proposed test amplification approach (Section 5.5).

At the end, we conclude the thesis in Chapter 6 with a discussion on possible future research directions.

[1] www.lowcomote.eu, This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement n°813884.

1.7 Scientific Production

The outcome of this thesis is published in two journals, two conferences, and one workshop.

— International journal

1. **Faezeh Khorram**, Erwan Bousse, Jean-Marie Mottu, Gerson Sunyé. Advanced Testing and Debugging Support for Reactive Executable DSLs. Software and Systems Modeling (2022).

2. **Faezeh Khorram**, Erwan Bousse, Jean-Marie Mottu, Gerson Sunyé. Adapting TDL to Provide Testing Support for Executable DSLs. The Journal of Object Technology, 20(3), pp.6:1-15, 2021.

— International conferences

1. **Faezeh Khorram**, Erwan Bousse, Antonio Garmendía, Jean-Marie Mottu, Gerson Sunyé, Manuel Wimmer. From Coverage Computation to Fault Localization: A Generic Framework for Domain-Specific Languages. Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE), 2022.

2. **Faezeh Khorram**, Erwan Bousse, Jean-Marie Mottu, Gerson Sunyé, Pablo Gómez-Abajo, Pablo C. Cañizares, Esther Guerra, Juan de Lara. Automatic Test Amplification for Executable Models. Proceedings of the ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS), 2022.

— International workshops

1. **Faezeh Khorram**, Jean-Marie Mottu, Gerson Sunyé. Challenges & Opportunities in Low-Code Testing. Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, 2020, Virtual.

Chapter 2

Background & State of the art

This section starts with a brief introduction on Model-Driven Engineering (MDE) (Section 2.1) and provides background of the DSLs considered in the scope of this thesis (Section 2.2). Meanwhile, we introduce a running example that will be used across this thesis. We then provide the testing background and state-of-the-art that is needed to understand the contributions of this thesis (Section 2.3).

2.1 Model-Driven Engineering

MDE is a software development paradigm that makes use of *models* as the pivotal development artifacts in order to manage complexity through abstraction [125]. A model provides a representation of some aspect of a system for a specific objective, such as providing a human understandable vision of some aspect of a system to involve the non-technical domain experts actively in the development lifecycle [57].

To define models, specific languages are used which can be classified in two groups:

- *General-Purpose Languages (GPLs)*: the languages that cover a broad range of concerns and can be used to model many aspects of a system. For example, the Unified Modeling Language (UML) has been proposed by the Object Management Group (OMG) [116].

- *Domain-Specific Languages (DSLs)*: the languages specialized for a particular application domain, hence enabling domain experts to create a system using the concepts they are familiar with [56]. For example, Business Process Model and Notation (BPMN) is a DSL for modeling business processes [112]. The person who defines a DSL is often called *language engineer* and the language user who defines models using the DSL is referred to as *domain expert*.

As DSLs are tailored to be used by domain experts, MDE processes usually rely on them [88, 104]. To build a system following the MDE principles, a domain expert first models the application using DSLs. Afterwards, the models are usually automatically transformed to either intermediate models (through model-to-model transformation) or source code (through model-to-text transformation), resulting in a deployable system. Abstraction along with automation provides simplicity, reusability, higher accuracy, complexity management, lower cost, and faster release time [35].

2.2 Domain-Specific Languages (DSLs)

A DSL is defined by a *syntax*, specifying what can be modeled using the DSL, and a *semantics*, defining the meaning of syntax constituents. More precisely, the syntax part comprises an *abstract syntax*, determining the concepts of a particular domain along with their relationships, and a *concrete syntax* specifying a representation for the abstract syntax elements to be used by the domain expert (e. g., providing graphical or textual symbols). Therefore, the semantics essentially provides meaning of the abstract syntax. Please note that the concrete syntax is left aside from this thesis.

2.2.1 Running Example: Arduino DSL

This thesis uses a sample DSL designed for modeling Arduino boards along with their behaviors as a running example. Arduino[1] is an open-source company that offers hardware boards with embedded CPUs, and with different modules (e. g., sensors, LEDs, actuators) that can be attached to a board. An Integrated Development Environment (IDE) is available to develop programs (called sketches) for such boards in C or C++. However, a DSL specifically defined for Arduino would help in developing the Arduino programs using required concepts rather than technical C instructions. In subsequent sections, we present the definition and the usage of an Arduino DSL.

2.2.2 Abstract Syntax

The *abstract syntax* of a DSL is usually defined as a *metamodel*[2]. Generally, a metamodel is made of a set of metaclasses, each containing a set of features. A feature can be either an attribute typed by a primitive type or a reference to

[1] https://www.arduino.cc/

[2] There are also other ways of defining an abstract syntax such as using grammars, but this thesis focuses on metamodels.

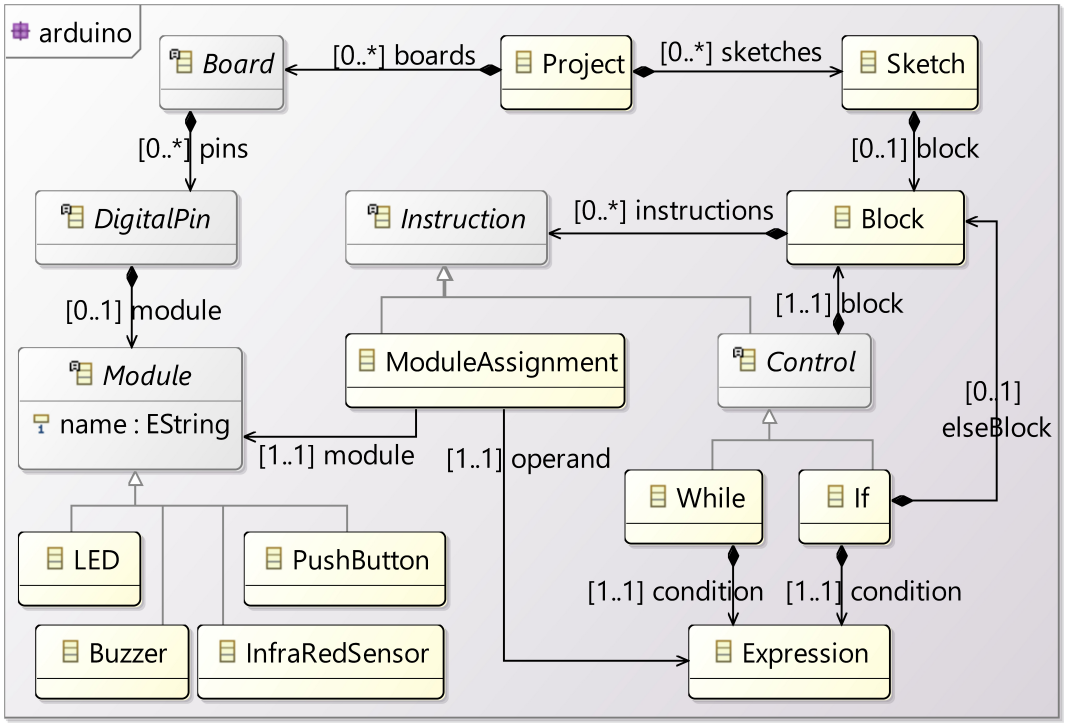


Figure 2.1: An excerpt of the abstract syntax of an Arduino DSL

another metaclass. Moreover, a metamodel possesses a *static semantics*, which is a set of structural constraints that must be satisfied by conforming models, such as multiplicities, containment references, or more complex constraints. There are specific languages to define metamodels and this thesis supports OMG's Meta-Object Facility (MOF) [113] and Ecore [132].

Example: Arduino abstract syntax. Figure 2.1 shows an excerpt of the abstract syntax of an Arduino DSL[3] as a metamodel. The root element is a Project which may contain several Board and Sketch elements. A Board represents an Arduino physical board. It contains several DigitalPin elements, each associated with one Module, such as LED, InfraRedSensor, PushButton, and Buzzer. The intended behavior of the boards must be defined using Sketch elements. A Sketch may contain a Block that may comprise several Instructions such as ModuleAssignment for changing the state of a Module and Control instructions to define conditional behaviors (e. g., using If or While).

[3]Inspired from https://github.com/mbats/arduino

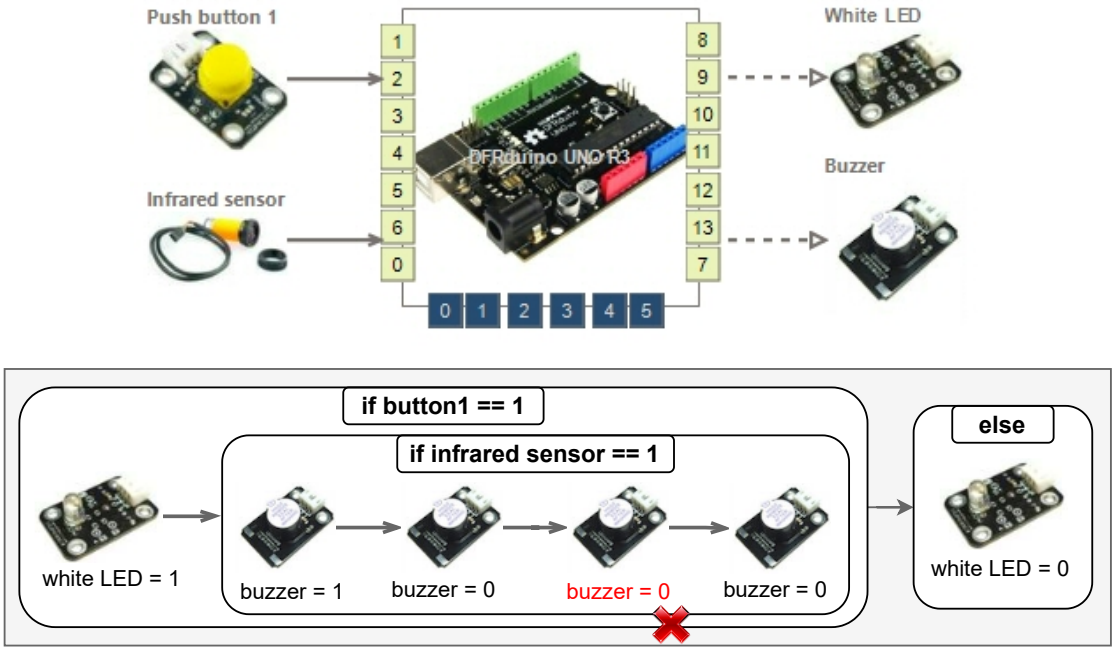


Figure 2.2: An example of an Arduino model representing a basic intrusion alarm system. It has a defect since the buzzer is not ringing as expected when the sensor detects an obstacle (it is highlighted in red where `buzzer` is mistakenly set to 0)

Modeling. Considering a DSL's abstract syntax as a metamodel, a model can be defined by instantiating the metaclasses of the metamodel and specifying values for their features to satisfy the static semantics—it is similar to the instantiation concept in Object-Oriented Programming (OOP) Languages such as Java. This is commonly referred as the conformity relationship between a model and its metamodel.

'The board's behavior, modeled on the bottom of Figure 2.2 using a Sketch element, is: "`if button1` is pressed, the `white LED` turns on and then if the `infrared sensor` detects an obstacle, the `buzzer` alternates between noise/silence periods twice (i. e., reporting an intrusion). Otherwise, the `white LED` turns off". Note that turning on/off an LED and a buzzer—the 'white LED=1', 'buzzer=1', 'white LED=0', and 'buzzer=0' in Figure 2.2—are indeed instances of Module-Assignment. We intentionally inject a defect in this model where `buzzer` should be set to 1 but it is mistakenly set to 0, meaning that the buzzer turns on but does not alternate between noise/silence states (highlighted in red in Figure 2.2). We aim to detect this defect with a test suite written and executed using our proposed approach.

2.2.3 Execution Semantics & Operational Semantics

The engineering of each DSL is commonly supplemented with a modeling environment that offers several services to assist the domain expert in modeling. Using the abstract syntax of a DSL, the domain experts can model particular aspects of a system. Among the existing DSLs, a large portion of them enable modeling the *dynamic* aspects of systems and the resulting models are so-called *behavioral* models (e. g., state machines [114], activity diagrams [115], and process models [24, 111]). If it was possible to execute behavioral models, the modeling environment could also provide dynamic Verification and Validation (V&V) techniques, such as debugging and testing. As a result, the domain expert could then analyze the behavioral models as early as possible and ensure the correctness of the modeled behavior early in the design phase.

To offer these facilities for a DSL, the DSL must provide an *execution semantics* which defines how to run its conforming models. In general, two main approaches are used for the definition of execution semantics:

- *Translational Semantics*: considering a target executable DSL, it transforms the models to target executable models or code using a *compiler*.
- *Operational Semantics*: it defines what are the possible runtime states of a model under execution and how such a runtime state changes over time to run the models directly through an *interpreter*.

A DSL with an execution semantics is called an *executable DSL (xDSL)* and its conforming models are referred to as *executable models (xModels)*. Our objective is to provide model-level testing support in order to perform early dynamic V&V using the domain concepts so as to be able to detect faults as soon as possible. This is achievable when the model can be executed by itself, hence for DSLs with operational semantics. Accordingly, this thesis focuses on those DSLs and thereafter, the term xDSL only refers to DSLs with operational semantics. Moreover, the term "model" from now on refers to an executable model.

The operational semantics of an xDSL comprises two parts: (*i*) the definition of the possible runtime states of a model under execution; and (*ii*) a set of execution rules defining how such a runtime state changes over time.

Runtime State Definition. To define the runtime state, several approaches are introduced so far that are all based on *extending* the abstract syntax [22, 68, 69, 78, 102, 130]. In this thesis, we consider the runtime state to be defined in a separate metamodel that introduces new classes and/or new features for the existing classes of the abstract syntax (later referred to as dynamic features)[4]. In

[4]The other ways to define the runtime state include using imports or inheritance relationships.

the literature, such a metamodel is referred to as *dynamic metamodel* [68, 69] or *runtime metamodel* [22, 130]. Extending the abstract syntax with the dynamic metamodel is performed by a non-intrusive extension mechanism, such as the UML package merge [116] in which two metamodels are merged by combining their classes; if two classes have the same name, they are combined in a class containing the properties from both originating classes. The metamodel resulting from merging the dynamic metamodel to the abstract syntax is called *execution metamodel* [30].

Execution Rules Definition. The execution of a model can be driven by changing its runtime state over time which is usually performed by a model transformation. A model transformation is usually defined at the metamodel level and comprises a set of rules, each one defining a subset of changes from a source to a target metamodel. When the source and the target metamodels are different, the transformation is called *exogenous* and otherwise *endogenous*. In general, by executing a model transformation on one or several source models (i. e., conforming to its source metamodel), one or several target models (i. e., conforming to its target metamodel) will be generated. However, there are endogenous transformations that do not create new target models but directly modify the source models which are named *in-place*.

To define a model transformation rule, two types of model transformation languages may be used:

- *Declarative languages*: each model transformation rule is composed of a source and a target pattern. When the source pattern can be found in a source model, it will be transformed as specified by the target pattern (e. g., ATL [82] or VIATRA [40]).
- *Imperative languages*: model transformation rules are indeed referred to as *operations*, each containing a sequence of statements altogether performing a change on the target model. There is usually one entry point operation that is called to start the transformation and operations may call each other, specifying the order of their application (e. g., Kermeta [78]).

The execution rules of an operational semantics are usually defined as an in-place endogenous transformation whose input is a model conforming to the execution metamodel that is modified during the execution by applying the transformation rules; the transformation is endogenous so its source and target metamodels are the same (i. e., the execution metamodel) and it is in-place, so the input model is directly modified.

In general, for every class of the xDSL's abstract syntax that has a runtime behavior, one execution rule is defined to implement such behavior. An execution

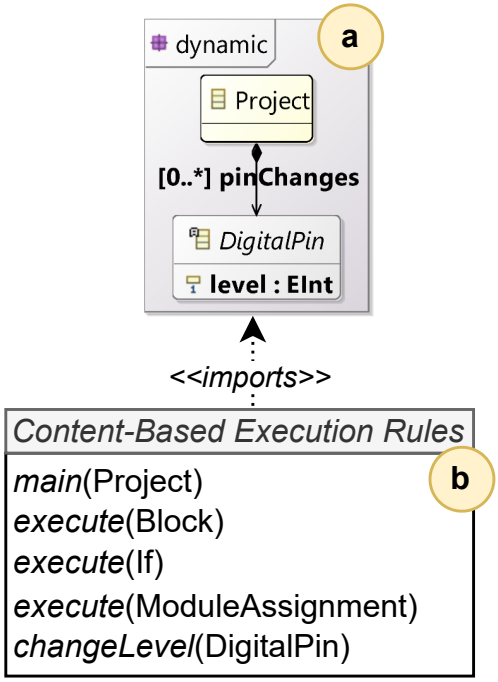


Figure 2.3: Definition of runtime state and execution rules for a content-based semantics of the Arduino DSL

rule that performs a change on a running model in according to the execution metamodel is called a *step rule* and an *execution step* is the application of a step rule (Section 2.2.4 on page 21 will provide more details).

In this thesis, we only consider xDSLs with discrete-event operational semantics (i. e., not continuous). Generally, these semantics can be defined as content-based or event-driven [93]. The former kind executes a model using an initial runtime state for the model that must be provided before the execution starts. The latter kind runs a model through an environment able to interact with the model execution using event occurrences. In the following, we clarify their differences.

2.2.3.1 Content-based operational semantics

A content-based semantics defines how to run a model in a *closed* environment, where only an initial runtime state is provided to the model before it is started. The execution rules of a content-based semantics comprise at least one rule acting as a starting point for the model execution, usually called the main(). This rule can trigger other execution rules (if any), and each may call other rules and perform observable execution steps in order to finalize the execution.

Content-based semantics of Arduino DSL. Figure 2.3 shows a content-based semantics for the Arduino DSL, comprising the runtime state definition (the dynamic metamodel in part (**a**)) and an excerpt of the execution rules (part (**b**)). The dynamic metamodel (part (**a**)) introduces new features for the metaclasses of the Arduino metamodel (already shown in Figure 2.1 on page 11) and will be merged into it. The DigitalPin has a dynamic feature named level which represents the state of its Module. For example, pressing a button, sensing an obstacle by a sensor, turning on an LED and a buzzer means the level of their DigitalPin is equal to 1. Thus to run the Arduino model of Figure 2.2 on page 12 using this semantics, we should set the level of the pins we want to be enabled to 1, before the execution. For example, to satisfy the if condition of the `sketch` part, the button must be pressed, so we should set the `level` of `button1` to 1 before execution starts. Also, to keep the track of changes of the level of DigitalPin elements during execution, we defined the pinChanges dynamic feature that is an ordered list.

Listing 2.1 on page 17 shows the implementation of the execution rules (part (**b**) of Figure 2.3 on page 15), written in Kermeta [78]. This language uses *aspect weaving* to extend a class with new features and/or execution facilities. The entry point main() rule (line 4)—that must be annotated with @Main as in line 3—is defined for the Project class (line 1). For each Sketch of a Project, it calls the execute() rule for its containing block (defined in line 10) which itself runs the instructions of the block in order.

According to the Arduino metamodel (Figure 2.1 on page 11), If conditions are a kind of Instruction. Such inheritance relationships can also be used when defining execution rules. For example, as can be seen in line 14, the If_Aspect extends the Instruction_Aspect and overrides its execute() rule (line 18). Note that a specific annotation namely @OverrideAspectMethod must be used to specify overriding (line 17). Moreover, the @Step annotation is required to distinguish a *step* execution rule from others (such as line 16).

For example, when the execution of the Arduino model of Figure 2.2 on page 12 starts, first the main() rule is called on the root Project object. This results in calling the execute(block) and then the execute(if button1==1) (i. e., implemented in line 18). As the level of the `button1` is 1, the `condition` of the `if` is satisfied, so the body of the `if` that is a Block is executed. As can be seen on the bottom of Figure 2.2, the body of the first `if` statement starts with a ModuleAssignment. Therefore, according to listing 2.1 on page 17, the execute() rule of the ModuleAssignment_Aspect (line 28) is called which turns on the white LED because the current value of its pin (= 0) is different from the value i. e., asked to be assigned (= 1). More precisely, the changeLevel rule of the DigitalPin_Aspect (line 37) is called in line 33 which adds a copy of the white LED to the list of pinChanges (line 40).

```
@Aspect(className=Project)
class Project_Aspect {
    @Main
    def void main() {
        _self.sketches.forEach[s|s.block.execute]
    }
}
@Aspect(className=Block)
class Block_Aspect {
    def void execute() {
        _self.instructions.forEach[i|i.execute]
    }
}
@Aspect(className=If)
class If_Aspect extends Instruction_Aspect {
    @Step
    @OverrideAspectMethod
    def void execute() {
        if (_self.condition.isSatisfied()){
            _self.block.execute()
        }
    }
}
@Aspect(className=ModuleAssignment)
class ModuleAssignment_Aspect extends Instruction_Aspect{
    @Step
    @OverrideAspectMethod
    def void execute() {
        val DigitalPin pin = _self.module.eContainer
        val previousValue = pin.level
        pin.level = _self.operand.evaluate
        if (pin.level != previousValue){
            pin.changeLevel
        }
    }
}
@Aspect(className=DigitalPin)
class DigitalPin_Aspect {
    @Step
    def void changeLevel(){
        _self.getProject.pinChanges.add(EcoreUtil.copy(_self))
    }
}
```

Listing 2.1: An excerpt of the content-based execution rules for the Arduino DSL, written in Kermeta

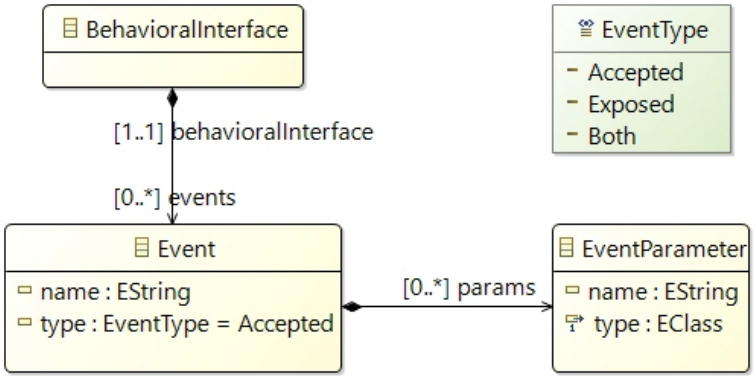


Figure 2.4: Behavioral interface metamodel [93]

2.2.3.2 Event-driven operational semantics

Although it is possible to execute a model solely based on its content, there are many cases requiring dynamically interacting with a running model, e. g., for running a co-simulation with other models [37, 93]. This requires the xDSL's operational semantics to have a real *event-driven* behavior that precisely specifies how one can interact with a running model, and how the said model should react. In this thesis, we consider that this aspect is handled by a language component called a *behavioral interface*, which we introduce in the next section as the foundation for the *event-driven semantics* of an xDSL. In the remainder of the thesis, xDSLs with content-based semantics are called *non-reactive* xDSLs, while xDSLs with event-driven semantics are called *reactive* xDSLs.

2.2.3.3 Behavioral interface

The behavioral interface of an xDSL specifies the types of events that are sent to and received from conforming models during their execution. It must be implemented by the execution rules of xDSL's operational semantics. While different approaches can be used to define such an interface (e. g., [44], [93]), this thesis uses the metalanguage proposed by Leroy et al. [93] whose concepts are presented in Figure 2.4. This metalanguage specifies a BehavioralInterface as a set of Events:

- Accepted events: specify what can be accepted by a running model.
- Exposed events: determines the observable reactions of a running model.

Each Event can have a set of parameters. The type of an EventParameter is indeed a metaclass of the xDSL's abstract syntax.

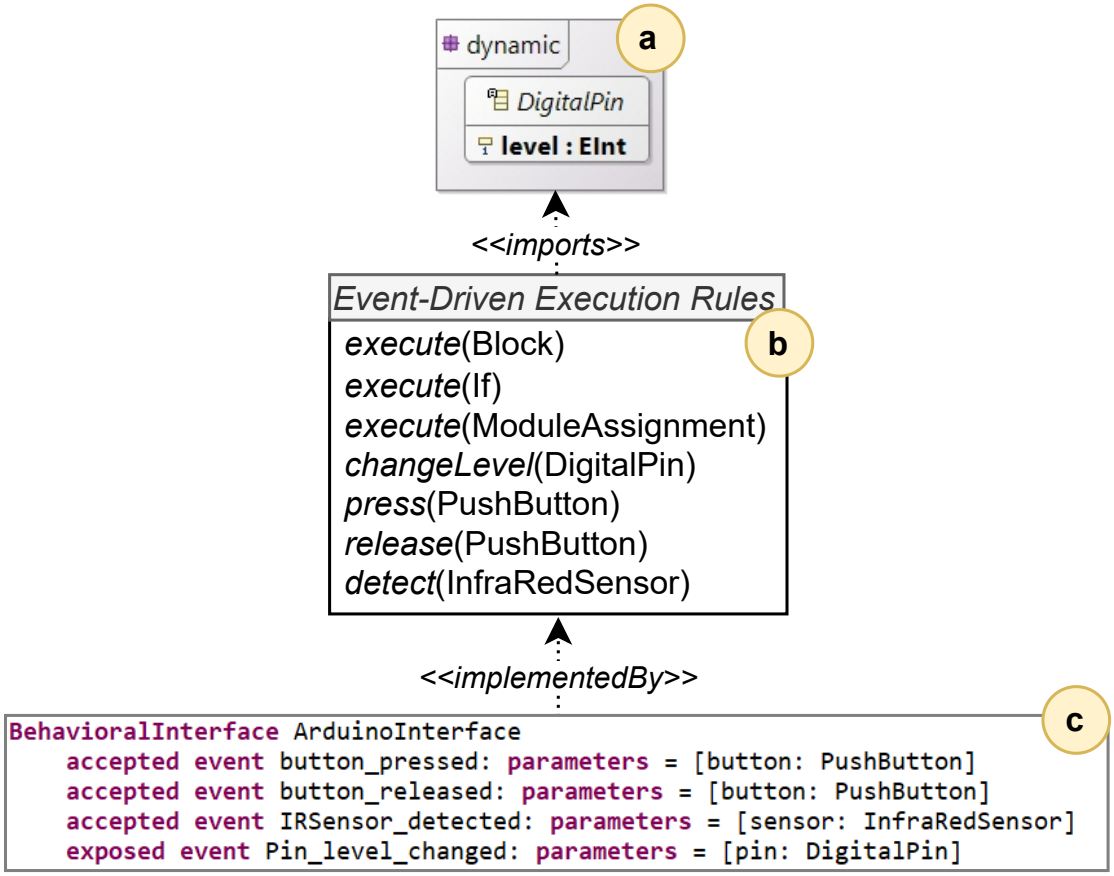


Figure 2.5: Definition of runtime state, execution rules, and behavioral interface for an event-driven semantics of the Arduino DSL

Event-Driven Semantics of Arduino DSL. Figure 2.5 presents an event-driven semantics for the Arduino DSL: the runtime state definition (part (**a**)), an excerpt of the execution rules (part (**b**)), and a behavioral interface (part (**c**)). The Kermeta implementation of the first three execution rules (i. e., execute (Block), execute (If), and execute (ModuleAssignment)) are the same as the ones shown in lines 8-36 of Listing 2.1 on page 17, and the rest (i. e., changeLevel (DigitalPin), press (PushButton), release (PushButton), and detect (InfraRedSensor)) are presented in lines 2-22 of Listing 2.2 on page 20.

The behavioral interface (Figure 2.5(**c**) on page 19) comprises four events, all implemented by the execution rules (Figure 2.5(**b**)):

- accepted event *button_pressed*: requests for pressing a button (implemented by press(PushButton) rule (line 4)).
- accepted event *button_released*: requests for releasing a button (implemented

```
1  @Aspect(className=PushButton)
2  class PushButtonAspect{
3      @Step
4      def void press() {
5          _self.pin.level = 1
6          _self.project.sketches.forEach[s|s.block.execute]
7      }
8      @Step
9      def void release() {
10         _self.pin.level = 0
11         _self.project.sketches.forEach[s|s.block.execute]
12     }
13 }
14 @Aspect(className=InfraRedSensor)
15 class InfraRedSensorAspect{
16     @Step
17     def void detect(){
18         _self.pin.level = 1
19         _self.project.sketches.forEach[s|s.block.execute]
20     }
21 }
22 @Aspect(className=DigitalPin)
23 class DigitalPin_Aspect {
24     @Step
25     def void changeLevel(){
26     }
27 }
```

Listing 2.2: An excerpt of event-driven execution rules for Arduino DSL, written in Kermeta

by release(PushButton) rule (line 9)).

- accepted event *IRSensor_detected*: requests for detecting an obstacle by a sensor (implemented by detect(InfraRedSensor) rule (line 17)).
- exposed event *pin_level_changed*: notifies changes of the `level` of the DigitalPin elements (implemented by changeLevel(DigitalPin) rule (line 22)).

For example, to run the Arduino model of Figure 2.2 on page 12 using this event-driven semantics, the occurrences of the Arduino's behavioral interface events must be communicated with the model during its execution. One can send a press event for the button1 which resulted in calling the press(PushButton) execution rule (line 4). It executes the block of each Sketch and similarly to what we described earlier in Section 2.2.3.1 on page 15 with content-based semantics of the Arduino DSL, the condition of the first if statement will be satisfied, so the white LED will

be turned on. This results in calling the changeLevel() rule (line 25) and is exposed by an occurrence of the pin_level_changed event.

The definition of a semantics for the Arduino DSL makes this language executable. From now on, we refer this Arduino xDSL as xArduino. Figure 2.6 shows its complete definition: its abstract syntax (a), its content-based semantics ((b.1) and (b.2)), and its event-driven semantics ((c.1), (c.2), and (c.3)).

2.2.4 Model Execution Tracing

A model execution trace tells what happened during the execution of the model, usually in a form of a sequence of information that is captured during the execution [72]. Such information could vary for different contexts and needs, and could be composed of various pieces of information, each related to the execution from a specific point of view. Among different existing definitions for model execution traces [72], this thesis relies on the one provided by Bousse et al. [31, 32]: an execution trace is a sequence of following information about a specific execution:

- execution states reached during the execution;
- changes made to the execution state of the executed model, such as the change in a value of a dynamic property, or the creation of a dynamic object;
- input and output event occurrences;

More precisely, we explained earlier that a model execution is driven by making calls to the execution rules of an xDSL operational semantics on the objects of the model, and this resulted in changes of the model's runtime state (i. e., changes of the value of its dynamic properties). Therefore, considering an executed model that conforms to an xDSL, the model execution trace specifies which execution rules of the xDSL's semantics were called by which elements of the model on which runtime state, as well as keeping the calls sequence (as shown in Figure 2.7).

Particularly for the reactive xDSLs, i. e., xDSLs with a behavioral interface, the execution trace of their conforming models also contains the occurrences of both accepted and exposed events.

For example, in Sections 2.2.3.1 and 2.2.3.2 on pages 15 and 18, we described a sample execution of the Arduino model of Figure 2.2 with non-reactive and reactive xArduino, respectively. The generated execution trace for each execution is:

- generated by non-reactive execution: execute (project (pinChanges = {})) » execute (block) » execute (if)» execute (White LED = 1) » changeLevel (White LED (level=1))
- generated by reactive execution: button_pressed (button1 (level=0)) » press (button1 (level=0)) » execute(block) » execute(if)» execute(White LED =

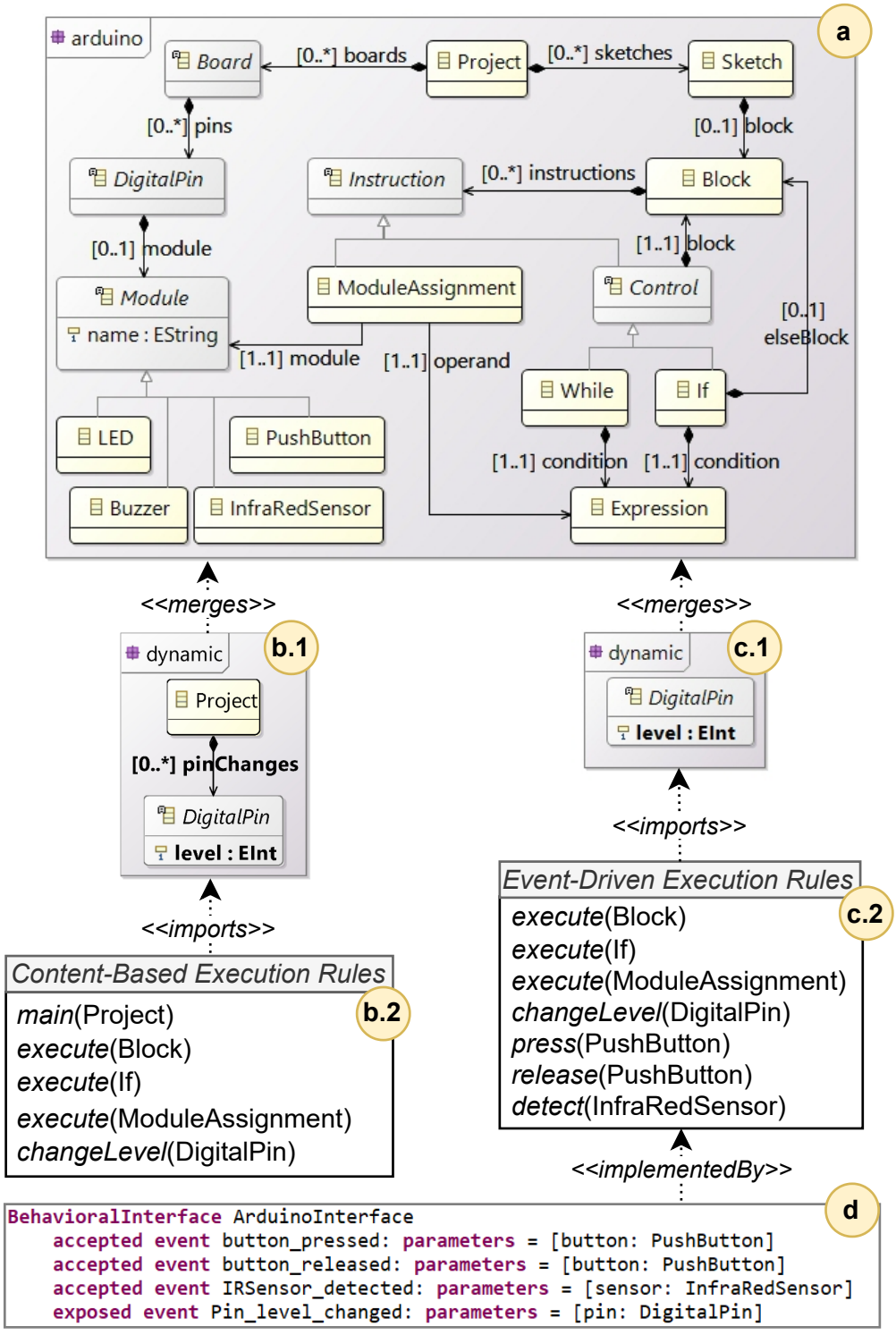


Figure 2.6: An overview of the Arduino xDSL definition

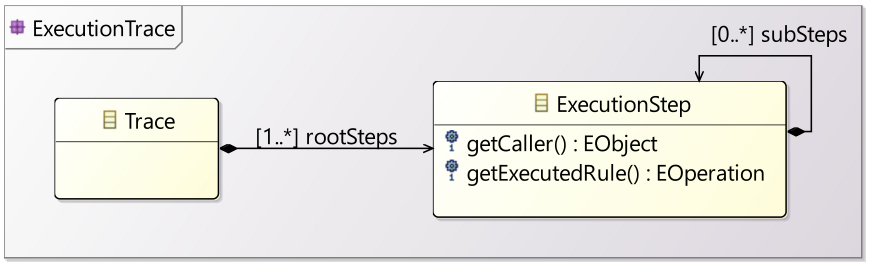


Figure 2.7: An excerpt of the execution trace metamodel [31, 32]

1) » changeLevel(White LED (level=1)) » pin_level_changed (White LED (level=1))

2.3 Testing

In this section, we provide the background and the state-of-the-art for testing models in general (Sections 2.3.1-2.3.3 on pages 23-28) and also for specific testing techniques supported by our proposed testing framework, including coverage computation (Section 2.3.4.1 on page 32), mutation analysis (Section 2.3.4.2 on page 34), interactive debugging (Section 2.3.5.1 on page 36), fault localization (Section 2.3.5 on page 35), and test amplification (Section 2.3.6 on page 39).

2.3.1 Terminologies

In this section, we define a number of terms that will be used later in this thesis. They are general terms of software testing mainly taken from the Ammann et al. book [12] and are adapted to the context of this thesis when needed.

Verification vs Validation. First of all, it is important to distinguish verification from validation:

- *Verification*: it evaluates whether individual parts of the system behaves correctly regarding the requirements and specifications.
- *Validation*: it evaluates whether the final system behaves as intended.

Testing can be done for verification (e. g., unit testing, integration testing) or validation (e. g., acceptance testing). In this thesis, our focus is on testing for verification of behavioral models.

Fault vs Failure. A *fault* is a static defect in the software that usually originated from design or implementation mistakes. A *failure* refers to the incorrect behavior of the software with respect to its expected behavior. Therefore, faults are the cause of failures.

Testing vs Debugging. The difference between fault and failure determines the difference between testing and debugging. *Testing* involves executing systems and observing whether they behave as expected or expose a failure. In case of a test failure, *debugging* helps to find the faults causing the failure.

Test Case, Test Data, Test Oracle, Test Suite. A *test case* is a set of all necessary information for a single complete execution and evaluation of the system under test. In general, a test case involves two kinds of data:

- *input data*: the input values that are required for the initialization and the execution of the system under test.
- *expected output data*: the result that must be produced by the system under test if it behaves as expected by the test case.

Moreover, a *test oracle* is a part of a test case that controls if the system behaves correctly. It verifies actual outputs against the expected output data e. g., checks the intermediate states, or the events exposed by the system under test. Also, a set of test cases are commonly referred to as a *test suite*.

Test Engineer. Defining test cases for a system requires knowledge about the system's behavior, so usually the designer of the system is a good person for defining test cases [12]. In software testing area, the system under test is a program implemented by a technical developer using a programming language. Accordingly, a *test engineer* is usually a technical expert who performs several tasks:

- defining test input data, expected output data, and test oracles,
- producing executable test cases and running them on the system under test,
- analyzing test results and determining if there is a fault in the system,
- reporting results to developers who are in charge of debugging the system.

In the context of this thesis, the system under test is indeed a behavioral model defined by a *domain expert*. One of our objectives is to enable the *domain experts* to perform early testing of the models they defined. Therefore, a *test engineer* is also a *domain expert* in the scope of this thesis.

Executable Test Case. An executable test case (also called executable test script [12]) performs a set of tasks automatically including, running the system under test with the test input data, getting the results produced by the system, comparing the results with the expected output, and preparing a clear report. Therefore, executable test cases increase the automation and usually reduce the costs, hence test engineers try to automate as many test cases as possible. This thesis aims at providing facilities for domain experts to define executable test cases for behavioral models.

Testing Framework & Testing Language. Executable test cases may be defined as Unix shell scripts, input files, or through a specific tool that is able to control the execution of the system under test. We call *testing framework* a tool that provides several facilities for test engineers:

- A software library to write executable test cases,
- A test runner to execute test cases,
- Supplementary services to analyze the test execution results.

We refer the language used for the specification of the testing library and the test cases as the *testing language*. A *testing language* could be either the same as the language used for the definition of the system under test or different from it. Therefore, it could be a programming language such as Java that is used in the JUnit testing framework, or a test-specific language such as Gherkin that is introduced by the Cucumber testing studio[5], or the standard Test Description Language (TDL) and Testing and Test Control Notation version 3 (TTCN-3) introduced by the European Telecommunications Standards Institute (ETSI) [63, 98]. This thesis uses TDL as the testing language of the proposed testing framework, introduced in the following.

2.3.2 Test Description Language

The Test Description Language was introduced by the European Telecommunications Standards Institute as a generic language for describing test cases. It aims at filling the gap between the abstract test requirements and the complex code of executable test cases to provide a common understanding of test cases for different stakeholders. TDL supports describing test objectives derived from system requirements and defining test cases that refine those objectives [98]. The standard semantics of TDL provides a loose semantics written in natural language [52] and a precise translational semantics using TTCN-3 as a target language [53] which is also standardized by the ETSI. A reference implementation of TDL is also provided,

[5] https://cucumber.io/docs/gherkin/

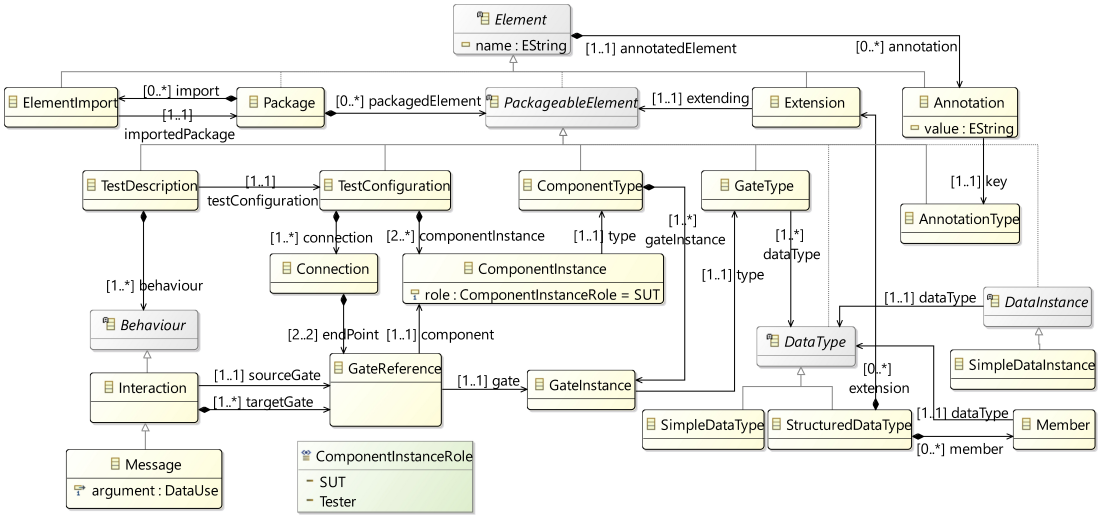


Figure 2.8: An excerpt of the TDL metamodel [52]

containing a standard abstract syntax, textual and graphical concrete syntax, and tools for model validation, and transformation to TTCN-3, among others. It is built atop the Eclipse Modeling Framework (EMF) and is available as an open-source project[6]

Figure 2.8 on page 26 presents the main elements of the TDL abstract syntax as a metamodel, and listing 2.3 on page 27 shows a conforming sample TDL model[7] written in TDL textual concrete syntax. A Package is the root element of a TDL model, hence the container of all other elements (such as `tdlModel` in line 1). To define a complete test case, three main pieces of information are required:

TDL Test Data. The first step in defining test data is to determine the required data types. TDL does not provide any concrete data type since its main objective is to be generic and platform-independent. So the testers should define their required types using the DataType element (such as `Page_Request` and `Page_Response` in lines 3 and 4), then instantiating them using DataInstance to define test data, both the input data that will be sent to the System Under Test (SUT) during test case execution (such as `not_existent_page_request` in line 21), and the expected output data that will be used to define the oracle of the test case (such as `error_404` in line 22).

TDL Test Configuration. A test configuration specifies a communication protocol between the test suite (later referred to as the *test system*) and the SUT.

[6] https://labs.etsi.org/rep/top/ide.
[7] taken from TDL website: https://tdl.etsi.org/

```
1  Package tdlModel{
2      //data types
3      Type Page_Request;
4      Type Page_Response;
5
6      //test configuration
7      Test Configuration web_conf{
8          create Tester test_browser of type Web_Browser;
9          create SUT tdl_website of type Web_Server;
10         connect test_browser.socket to tdl_website.socket;
11     }
12     Component Type Web_Server having{
13         gate socket of type Web_Port;
14     }
15     Component Type Web_Browser having{
16         gate socket of type Web_Port;
17     }
18     Gate Type Web_Port accepts Page_Request, Page_Response;
19
20     //test data
21     Page_Request not_existent_page_request;
22     Page_Response error_404;
23
24     //test cases
25     Test Description page_not_found uses configuration web_conf {
26         test_browser.socket sends not_existent_page_request to
       tdl_website.socket;
27         tdl_website.socket sends error_404 to test_browser.socket;
28     }
29 }
```

Listing 2.3: An example TDL model (taken from TDL official website)

TDL follows a component-based approach, hence a TestConfiguration comprises two or more ComponentInstances (such as `web_conf` configuration in lines 7-11), one in the role of SUT (such as `tdl_website` in line 9) and the rest as Tester (such as `test_browser` in line 8). It also defines the Connections between the components (line 10). A ComponentInstance is typed by a ComponentType, which determines the component communication channels using the so-called gates (such as `Web_Server` and `Web_Browser` in lines 12-17). Accordingly, it contains at least one gate (i. e., GateInstance) that is instantiated from a GateType. A GateType defines what kind of data can be exchanged through its instances (such as `Web_Port` in line 18).

TDL Test Description. To describe the behavior of a test case, the TestDescription element should be instantiated. For example, `page_not_found` Test

Description in lines 25-28 checks whether `tdl_website` responds with a 404 (page not found) error, when it receives a request for a non existent page. It uses one of the previously defined TestConfiguration instances, and contains a sequence of Behavior elements. Currently, twenty types of behavior are defined in the TDL standard for describing expected behavior, failure upon deviations by default, actions and interactions, and alternative, parallel, iterative, conditional, interrupting, defaulting, and breaking behaviors. For example, lines 26 and 27 are instances of the Message behavior. When the Message is sent from the tester component to the SUT, the sent data is test input data (line 26) and otherwise is considered as an expected output and the Message is thus equivalent to an assertion.

Moreover, TDL allows (i) importing a Package into another Package (using ElementImport); (ii) extending a PackageableElement (for example, it is possible to define an inheritance relationship between StructuredDataTypes by defining Extension elements for them); and (iii) annotating Elements using Annotation.

2.3.2.1 TDL limitations

The presented example shows that TDL can be used to describe test cases in a high level of abstraction, hence it is a good fit to be used by domain experts but also challenging due to several reasons. First, all the required data types (e. g., lines 3–4) and test configurations (e. g., lines 7–11) have to be manually defined by the domain expert. However, this effort is costly and error-prone and the domain expert is unlikely to be knowledgeable enough for doing it.

Second, the TDL test cases are not directly executable. While a translational semantics using TTCN-3 as a target language exists for TDL [53], this semantics is only partial, and mainly aims to manage test cases for software systems communicating through common protocols (TCP, UDP, TELNET, SQL, HTTP, etc.). Therefore, to use TDL for testing executable models, a new execution semantics is required.

2.3.3 Testing Frameworks for DSLs

A testing framework for an xDSL enables testing the models conforming to the xDSL. In the literature, there are both testing frameworks for particular xDSLs and generic testing frameworks that are applicable to a wide range of xDSLs. In this section, we provide an overview of their state-of-the-art.

2.3.3.1 DSL-specific approaches

Mens et al. propose a specific methodology for designing and early testing executable statecharts [103]. In the design phase, several tasks have to be performed, some of

which are required for the testing phase such as defining the execution scenarios, implementing the mapping between the steps of the scenarios and the statechart test primitives (in Python), and writing unit tests (in Python). The scenarios and the unit tests will then be executed on the statecharts to verify their behavior. Although they support a complete process of designing and validating statecharts, the testing activities should be performed by a technical tester as coding in Python is required.

Iqbal et al. [75] aim at enabling the domain expert to perform testing of Real-Time and Embedded Systems (RTES) as well as reducing the cost of testing on real platforms. They propose the use of modeling languages—UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE) and Object Constraint Language (OCL)—for simulating the execution environment and the hardware platform, but not modeling the software system under test. Therefore, their approach improves the involvement of the domain experts in testing by allowing early testing of the real RTES systems in a simulated environment. Hili et al. propose an approach for interactive monitoring of real-time and embedded systems modeled using UML Real-Time (UML-RT) [71]. Their approach enables different external components such as tools for data collection, animation, simulation, analysis, adaptation, and control to monitor the execution of the code generated from a UML-RT model. As one of their case studies, they show how the approach can be used to observe the functional behavior of the generated code. Therefore, although they do not provide any testing approach for writing test cases, they show how the approach can be applied for testing purposes. Moreover, they are focusing on monitoring the behavior of the code generated from a model, not the executable model itself.

To tackle the inherent complexity of testing domain intensive cloud applications, a configurable test DSL is proposed by Santiago et al. [124]. Given an abstract definition for a cloud application (including its user interface, user interactions, data setup, environment, and platform configuration) using the domain concepts, it generates a specific test DSL and a testing toolset named Legend for authoring, executing, and debugging test cases for cloud applications [86].

In the context of measurement systems, a specific DSL named Sequencer is used in the NASA awarded measurement system (DEWESoft) which enables adjusting measurements and creating measurement procedures. To provide testing support for the Sequencer DSL, Kos et al. propose a specific testing framework named Sequencer Testing Tool (SeTT) [87]. The SeTT tool enables the domain expert to define test cases for each part of the measurement system. It indeed allows augmenting test elements such as assertions into the Sequencer models.

To define test cases for the executable business processes that are modeled using Web Services Business Process Execution Language (WS-BPEL) or BPMN2, Lübke

and Van Lesson propose a specific testing approach [97]. They use a metamodel extension technique to add test-specific elements (e. g., assertions) to the BPMN metamodel. To ensure the test models have deterministic behaviors, they enforce some control-flow restrictions. The domain expert can define test cases as BPMN models in which there is one *Pool* describing the process under test and other *Pools* specifying the test case behavior. The *Pools* communicate with each other by exchanging messages. To execute such test models, the technical information for running the physical operations of the process under test must be provided in advance.

A sizable amount of work is proposing testing approaches for fUML [115] which are described below. A Behavior-Driven Development (BDD) framework enables describing the requirements as executable user stories and the acceptance criteria as executable scenarios attached to the user stories. Lazăr et al. propose a BDD framework for fUML by defining a UML profile for BDD and a BDD library comprising executable commands required when describing fUML scenarios [91]. The framework allows the domain expert to define fUML models following a BDD approach, meaning that he/she first defines executable fUML stories and scenarios and then describes the fUML models satisfying them.

Arnaud et al. propose a testing approach for fUML where the behavioral scenarios of a system are first described using UML sequence diagrams enriched with timing properties that are described in the UML MARTE constraint language [18]. These diagrams describe the communications between the different components of a system, and each component is itself described using fUML activity diagram. In this work, a testing tool is provided which automatically evaluates the conformance of the fUML activities with the sequence diagrams and their timing constraints. In addition, they generate test input data from the sequence diagrams and use them to test the behavior of the activity diagrams automatically.

Mijatov et al. propose a functional testing framework to validate the behavior of fUML models [106]. For describing test cases, they provide an executable test specification language that supports using temporal expressions for the precise selection of the runtime states to be asserted, using OCL queries for specifying complex assertions on the runtime states of a system that behaves concurrently, and verifying the execution order of the activity nodes for concurrent systems.

Iqbal et al. introduce a simulation and test generation approach for the fUML activity diagrams containing Alf[8] code [74]. At first, the fUML models are translated into Java code. Afterward, the test input data is generated automatically from the Java code, enabling an exhaustive simulation of the fUML models. Finally, using the provided simulation, the test cases along with the test oracle are auto-generated satisfying 100 % coverage of the Java code.

[8] Action language for fUML

2.3.3.2 Generic approaches

When a grammar-based DSL has a translational semantics, if the target language (i. e., a General-Purpose Language (GPL)) provides a unit testing framework (e. g., JUnit for Java), then the work of Wu et al. provides a unit testing framework for that DSL [152]. It requires the language engineers to define the mapping algorithms between the testing actions of their DSL and the target GPL. Accordingly, the framework can translate test cases from DSL code to GPL which enables using the GPL testing tools for executing test cases on the generated GPL code of the model under test. It also translates the test results from the GPL code to the DSL, to report the result using the domain concepts. Therefore, this approach is useful for compiled DSLs and performs testing at the code level.

Sometimes, defining the expected output of a system for a given input data is not possible or is too expensive. This problem is referred to as *oracle problem* and to test systems with oracle problem, *metamorphic testing* technique is usually used. This technique tests a system based on the relationships between multiple inputs and their outputs. Cañizares et al. propose a generic approach for providing a metamorphic testing environment for a given xDSL automatically [36]. They offer a specific DSL namely Gotten to write metamorphic relations (i. e., test cases in the metamorphic testing approach). The language engineer must define how to run the metamorphic relations on a model and how to validate if a relation is satisfied.

The work of Meyers et al. is the closest to our work [105]. They propose a generic testing approach for xDSLs with discrete-event semantics i. e., reactive xDSLs. Given an input xDSL, it generates an xDSL-specific testing language by extending the abstract syntax of the xDSL with a limited set of testing features. To execute each test case written using this language, the operational semantics of the xDSL must be instrumented specifically for it. Instrumentation means new execution rules (i. e., for test case execution) must be added to the xDSL's execution rules. This in turn requires the language engineer to enrich the abstract syntax of the xDSL with event-related concepts to specify where new rules must be added.

2.3.3.3 Limitations

To sum up, many testing frameworks aim at providing testing facilities for behavioral models. Among them, some perform testing at the code level (i. e., the code generated from behavioral models) [71, 74, 103, 152], thus they are not usable by domain experts who do not have the technical knowledge to work with the generated code. In contrast, some allow testing at the model level by enabling the domain experts to describe test cases using the DSL they are familiar with [18, 86, 87, 91, 97, 106, 124]. Although they promote usability for the domain experts, they lack reusability since a new testing language must be engineered for each new

DSL, representing a costly and error-prone work.

To solve the reusability issue, more generic approaches have also been proposed which target a wide range of xDSLs. However, they expose particular constraints that make their use limited and/or difficult. More precisely, the work of Wu et al. [152] needs a bidirectional compiler between a given grammar-based compiled DSL and its target executable language, and the approach proposed by Cañizares et al. [36] is only applicable for metamorphic testing and does not provide any test execution facility by itself. Also about the work of Meyers et al. [105], as it requires changing the xDSL definition for each test case to be able to execute it, their approach is neither directly applicable to a given xDSL nor easily usable by the domain experts.

Lastly, all the existing testing frameworks in the context of xDSLs are mainly focused on providing test case definition and execution facilities. However, a proper testing framework must also support its users in performing post-execution activities, such as measuring quality, debugging in case of test failure, and improving when the quality is low. In the remainder of this chapter, we provide the background for each of these concerns because they will be discussed in this thesis.

2.3.4 Test Quality Measurement

The number of potential inputs for most programs is large and cannot be explicitly enumerated. Therefore, it is not always feasible to perform a *complete* testing of a program and there is a need for a stop criterion that determines how much testing is required to reach a certain level of confidence in the program's verification. Test quality measurement techniques aim at providing such a criterion. Measuring the quality of a test suite often helps to decide whether the test suite should be improved, and how much effort should be put into this endeavor. In the realm of programming languages, coverage computation and mutation analysis are two popular means of test quality measurement [12] which are introduced in this section.

2.3.4.1 Coverage computation

Coverage computation is a test quality measurement technique which analyzes how much of the system under test is exercised by a given test case based on a given criterion. There are many coverage metrics in the literature, and each of them observes the system execution from a different perspective. For example, in the context of programming languages, *statement coverage* metric computes the percentage of the statements of the software that are executed and *method coverage* metric calculates the percentage of the methods whose at least one of their inner statements is executed [12].

The most used approach for measuring coverage of software programs is *instrumentation* in which additional code that does not change the behavior of the program is added to the program to collect information about whether some requirement have been met. For example, listing 2.4 shows a Java program and listing 2.5 shows how it is instrumented to capture if the body of an if block has been reached during an execution (line 4 is added) [12].

```
1 public int main (A, B){
2     int m = A;
3     if (A>B){
4         m=B;
5     }
6     return (m);
7 }
```

Listing 2.4: Original function (taken from [12])

```
1 public int main (A, B){
2     int m = A;
3     if (A>B){
4         Mard: "if body has been reached"
5         m=B;
6     }
7     return (m);
8 }
```

Listing 2.5: With instrument (taken from [12])

Coverage Computation for programming languages. In the context of programming languages, several coverage frameworks are already proposed. For example, Misurda et al. [107] propose a tool called *Jazz* for testing Java programs and measuring their coverage using a dynamic instrumentation technique. This technique inserts and deletes instrumentation during the execution of the program as needed by the considered coverage metric. Bordin et al. [28] introduce the *Couverture* tool which is able to measure structural coverage from detailed execution traces produced by a virtualized execution platform. *CodeCover* is a well-known coverage tool proposed by Patil et al. [117] for Java and COBOL languages which supports several coverage metrics including, statement coverage, branch coverage, loop coverage, and condition/decision coverage. While most of the instrumentation-based tools work at the byte code level, meaning that they instrument the byte code generated from the source code, CodeCover instruments the source code itself to produce more accurate measurements.

Sakamoto et al. [123] propose an extensible tool called *Open Code Coverage Framework* (OCCF) aiming at providing coverage computation for those programs implemented by several programming languages, such as web applications. It is customizable for new programming languages and supports both a set of existing test coverage criteria and the addition of new developer-defined test coverage. Their evaluation on eight programming languages (C, C++, C#, Java, JavaScript, Python, Ruby and Lua) demonstrated the benefits of using generic approaches for computing coverage.

Coverage Computation for DSLs. The benefits of using coverage computation in the context of software testing motivated its usage in the context of model testing. Accordingly, several research efforts have proposed the use of existing coverage criteria for specific modelling languages, e. g., logic coverage for State Machines [50], data-flow coverage for executable UML models [145], branch coverage for activity diagrams [27], among many others. However, to the best of our knowledge, there is no generic coverage criteria for executable models. Also, this topic is not yet discussed within the context of language workbenches [51].

2.3.4.2 Mutation analysis

Mutation analysis is a popular test quality measurement technique, which follows this idea: if we inject artificial faults into a program, an existing test suite that can find these faults is probably good enough at discovering real faults [47]. The artificial faults are defined in the form of *mutation operators* which perform small modifications (e. g., flipping $>$ by $<$ in an expression or changing the value of constants) on the source code. They are systematically applied on a program to produce a set of mutants (i. e., faulty programs). Afterward, the program's test suite is run on each mutant. If there is at least one test case in the test suite that its execution is different for the program and the mutant, we conclude the test suite has detected the fault of the mutant, and the mutant is said to have been 'killed' by the test suite. Finally, a mutation score is calculated as follows:

$$mutation\ score = \frac{number\ of\ killed\ mutants}{number\ of\ generated\ mutants}$$

This mutation score is a criterion for measuring the quality of the test suite [13].

Mutation Analysis for programming languages & DSLs. Providing mutation analysis for a specific language requires:

- a way to define mutation operators for the language and apply them on the programs defined by the language to generate mutants.
- a way to execute programs' test suites on the mutants and calculate their mutation score.

Accordingly, we can find many efforts in the literature introducing mutation operators for specific programming languages (e. g., C [4, 80], C++ [46], Java [34, 85]) as well as modeling languages (e. g., UML class diagrams [64], UML state machines [7], finite state machines [95, 119], Petrinets [54]). The mutation analysis of model transformations, part of xDSLs, has also been considered. Mottu et al. [109] introduced a set of mutation operators dedicated to model transformation testing

but remaining language-independent. Their mutation operators are *metamodel-based*, considering the metamodels involved in a model transformation (cf page 14) instead of the transformation language. It has then been adapted to specific model transformation languages such as ATL (Guerra et al. [65]). The problem of generating model transformation mutants with a generic approach has also been studied by Aranega et al. [14]. It also investigated how to define mutation operators for any DSL in a systematic approach [9]. Gómez-Abajo et al. propose a specific language named WODEL which allows to define mutation operators for any metamodel-based DSL [61].

Moreover, several approaches and tools are proposed so far for the systematic application and execution of mutants and test suites, such as Proteum for finite state machines [55], MoMut for UML state machines [89], and WODEL-Test [62] that is a generic tool which automatically provides a mutation testing environment for any metamodel-based DSL if (*i*) the language engineer defines the mutation operators for his/her DSL using WODEL language; and (*ii*) there is an existing testing framework for the DSL which provides facilities for testing DSL's conforming models. This thesis uses WODEL-Test as it is a generic tool which can be applied on DSLs considered in the scope of this thesis (will be discussed in Section 4.3 on page 89).

For example, listing 2.6 shows a few of the xArduino mutation operators that we defined using the WODEL language [61] in the context of this thesis. It starts with specifying the mutant generation mode which could be a specific number or exhaustive to generate all the possible mutants (line 1). A WODEL file also needs to know the output path for saving the generated mutants (the in path in line 2), the path to the input model (the from path in line 3), and the path to the input metamodel (the metamodel path in line 4). The presented mutation operators act as follows:

- `cmar`: if the input model has at least one ModuleAssignment instance and at least two Module instances, it generates a mutant by changing the module reference of the ModuleAssignment to another Module.
- `cbbo_equal`: if the input model has at least one BinaryBooleanExpression instance that its operator is not 'equal', it generates a mutant by changing the operator to 'equal'.
- `cic`: if the input model has at least one IntegerConstant, it generates a mutant by changing the value of the constant to a random integer.

2.3.5 Fault Localization

When a test case fails, assuming the test case is correct, there is a fault in the system under test that must be fixed. To fix this fault, we need to first find its

```
generate exhaustive mutants
in "data/out/"
from "data/model/"
metamodel "/../arduino.ecore"

with blocks {
    cmar "Changes ModuleAssignment reference to another module" {
        m = select one Module
        modify one ModuleAssignment where {module <> m} with {module = m}
    }
    cbbo_equal "Changes Binary boolean expression operator: equal" {
        modify one BinaryBooleanExpression where {
            operator in ['sup', 'inf', 'infOrEqual', 'supOrEqual', 'AND',
     'OR', 'Different']
        } with {operator = 'equal'}
    }
    cic "Changes the value of integer constant"{
        modify one IntegerConstant with {value = random-int(2,4)}
    }
    ...
}
```

Listing 2.6: Some of the xArduno mutation operators implemented in WODEL

location and this can be performed using two sets of techniques [150]:

- *Traditional techniques*: they are usually applied for manual fault localization of small programs.
- *Advanced techniques*: they are commonly used for automatic fault localization of large and/or complex programs.

According to a recent survey on software fault localization techniques, *interactive debugging* is a popular traditional technique, and the mostly used advanced technique is Spectrum-Based Fault Localization (SBFL) [150]. This thesis considers these two approaches which are explained in the following.

2.3.5.1 Interactive debugging

Interactive debugging involves manual control and observation of an execution with the help of an interactive debugger. Such debugger provides services to *pause* and *unpause* the execution through *breakpoints*—i. e., conditions upon which the execution must be paused, such as "reaching a specific model element"—and prepares information to observe the execution, such as the current stack of method calls or the values of all existing variables [150]. An execution can be represented as

a sequence of execution steps (e. g., a sequence of statements), and a step may itself contain a sequence of inner steps (e. g., method calls, leading to more statements). Based on this representation, an interactive debugger also provides a common set of operators to perform step-by-step observation of an execution, such as:

- The *resume* operator, to continue the execution until a breakpoint is reached.
- The *step over* operator, to continue the execution until the end of the current step or until a breakpoint is reached, hence ignoring the possible inner steps.
- The *step into* operator, to continue the execution until either some inner step is reached (if any) or when the current step ends.

Note that a typical interactive debugger offers other services as well, such as conditional breakpoints or the ability to query/change the model runtime state. Yet, this thesis focuses only on the above-described stepping operators—which are the most essential services of an interactive debugger—and leaves other debugging services for future work.

Interactive Debugging for DSLs. Providing debugging facilities for domain-specific languages has been investigated by several works. In a nutshell, there are approaches for debugging both executable UML models [37] and executable EMF models [30]. According to a recent survey on UML model execution [37], 21 out of 82 solutions are concerned with providing debugging facilities at the model level, 13 of which offering tools as well. Bousse et al. [30] introduce a generic omniscient debugging approach for executable DSLs. More specifically, their approach can be configured for a given xDSL and then can be used to perform forward and backward debugging of its conforming executable models. In this thesis, we use this generic approach as it supports the xDSLs considered in the context of this thesis (i. e., xDSLs defined according to the definitions given in Section 2.2 on page 10). It is worth mentioning that, all the existing works in the context of model-level debugging are focusing on debugging one model at a time while this thesis is concerned with debugging two models—a test case and its model under test—interactively (will be discussed in Section 5.3 on page 102).

2.3.5.2 Spectrum-Based Fault Localization (SBFL)

SBFL is a popular advanced fault localization technique that uses the results of test cases and their corresponding code coverage information to estimate the likelihood of each program component of being faulty. Depending on how the coverage is computed (i. e., the coverage metric), the examined components could be different. For example, when SBFL uses statement coverage for a Java program, it calculates the probability of each program's statement being faulty [150]. Generally, each

SBFL technique introduces an arithmetic formula that is based on a set of values which are computed from the test verdict and coverage information:

- N_{CF}: number of failed test cases that cover a statement
- N_{UF}: number of failed test cases that do not cover a statement
- N_{CS}: number of successful test cases that cover a statement
- N_{US}: number of successful test cases that do not cover a statement
- N_C: total number of test cases that cover a statement
- N_U: total number of test cases that do not cover a statement
- N_S: total number of successful test cases
- N_F: total number of failed test cases

For example, a well-known formula is Tarantula [81], defined as:

$$(N_{CF}/N_F)/(N_{CF}/N_F + N_{CS}/N_S)$$

It follows the idea that the elements executed by more failed test cases are more likely to be faulty, and the ones executed by more passed test cases are less likely to have a fault. For instance, Table 2.1 shows an example of using the Tarantula SBFL technique to find the fault of a Java program (this sample is taken from reference [150]). The program (column 2) takes a natural number namely a as input. If it is greater than 1, then its addition to 1 and its multiplication by 2 will be printed. Otherwise, its minus from 1 and the number itself will be printed. The program has a bug in statement s_7 where the a number is not doubled. This should be noted that, as SBFL considers each line of the program as a statement, the way of writing the program may have a direct impact on the result produced by SBFL techniques.

Table 2.1 presents three test cases of the program (tc_1, tc_2, tc_3 in columns 3-5) along with their execution result in the last row; the first two test cases are passed (i. e., S for Successful) but the last one is failed (i. e., F for Failure). Moreover, the coverage status of each program statement by each test case is determined using the "•" symbol. Using these test verdict and coverage information, the values for N_{CF} and N_{CS} parameters are calculated. Considering statement s_7, one failed test case has covered this statement ($N_{CF} = 1$) and no passed test cases cover it ($N_{CS} = 0$). These values are then used for computing the suspiciousness score of each statement based on Tarantula technique. Afterward, the statements are ranked based on their score and the top-ranked ones are the most likely ones to be faulty. As can be seen, the statement s_7 is correctly ranked first.

SBFL for DSLs. Finding faulty elements of models has already been investigated in the literature. Wang et al. [146] propose the application of spectrum-based and mutation-based fault localization techniques for declarative models implemented

Table 2.1: An example showing the suspiciousness values computed using the Tarantula technique (taken from reference [150] © 2016 IEEE)

	Code with a bug at s_7	tc_1	tc_2	tc_3	N_{CF}	N_{CS}	Susp	Rank
s_1	input(a)	•	•	•	1	2	0.5	3
s_2	$i = 1$;	•	•	•	1	2	0.5	3
s_3	$sum = 0$;	•	•	•	1	2	0.5	3
s_4	$product = 1$;	•	•	•	1	2	0.5	3
s_5	if $(i < a)\{$	•	•	•	1	2	0.5	3
s_6	$sum = a + i$;			•	1	0	1	1
s_7	$product = a * i$; //**bug**			•	1	0	1	1
s_8	} else {	•	•		0	2	0	10
s_9	$sum = a - i$;	•	•		0	2	0	10
s_{10}	$product = a/i$;	•	•		0	2	0	10
s_{11}	}	•	•		0	2	0	10
s_{12}	print(sum);	•	•	•	1	2	0.5	3
s_{13}	print(product);	•	•	•	1	2	0.5	3
Test Execution Results		S	S	F				

in Alloy. Although BLiMEA [16] and Ebro [15] detect errors in models based on evolutionary algorithms and not SBFL, Arcega et al. [17] compare these proposed tools for bug localization and show that the combination of these tools outperforms existing approaches. Some studies detect the faulty element in model transformations based on SBFL [96, 142]. Troya et al. [142] present an approach to apply SBFL to locate the faulty rule in a model transformation and evaluate the effectiveness of their approach by comparing a large set of different state-of-the-art SBFL techniques, which is also reused in the context of our work. Nevertheless, to the best of our knowledge, there is no generic approach providing advanced fault localization facilities for xDSLs.

2.3.6 Test Amplification

Test amplification refers to all the existing techniques aiming at enhancing manually-written test cases based on a specific goal, such as improving the coverage of changes or increasing the accuracy of fault localization [42]. A subset of these techniques focuses on improving manually-written test cases to avoid regression faults. Given a test suite for a system, such techniques create new test cases by modifying the test input data of existing test cases, and then run the system with this modified data to put the system in unexplored states. For each new test case, an oracle is generated by inferring assertions from the resulting execution trace of the system.

As new test cases are based on the current behavior of the system, these techniques can effectively strengthen regression testing [43, 153].

For example, Listing 2.7 on page 40 shows a Java class named Stack which is a simple implementation of a stack that stores unique elements in the elems array. Using the push and pop methods, we can perform the standard push and pop operations on the stack. Also we can check if the stack is full or empty using the isFull and isEmpty methods, respectively. Listing 2.8 on page 40 shows a test case for the stack Java class that has no assertion. It might be generated by an automatic test case generation tool and is only useful to detect uncaught exceptions or violations of some predefined contracts. Using a test amplification approach, a test case like the one shown in Listing 2.9 on page 41 can be generated. As can be seen, comprehensive assertions based on the current version of the system under test are added to the initial test case. These assertions are indeed helpful in detecting regression faults introduced in future program versions.

```
1 public class Stack {
2     private Comparable [] elems;
3     public Stack() { ... }
4     public void push(Comparable i) { ... }
5     public void pop() { ... }
6     public boolean isFull () { ... }
7     public boolean isEmpty () { ... }
8 }
```

Listing 2.7: Example of a toy class (taken from [42])

```
1 public class StackTest {
2     @Test
3     public void test1 () {
4         Stack s1 = new Stack();
5         s1.push('a');
6         s1.pop();
7     }
8 }
```

Listing 2.8: An initial JUnit test case for the toy class of Listing 2.7 (taken from [42])

Since test amplification may generate large amounts of test cases, it is important to keep only the relevant ones. When the goal is to increase test case effectiveness in detecting regression faults, an efficient technique for identifying relevant test cases is mutation analysis, already described in Section 2.3.4.2 on page 34. We use mutation analysis to check the degree of improvement that test amplification provides, and as selection criterion for the most effective amplified test cases.

```
1  public class StackTest {
2      @Test
3      public void test1 () {
4          Stack s1 = new Stack();
5          assertTrue(s1.isEmpty());
6          assertFalse(s1.isFull());
7          s1.push('a');
8          assertFalse(s1.isEmpty());
9          assertFalse(s1.isFull());
10         s1.pop();
11     }
12 }
```

Listing 2.9: An augmented JUnit test case (taken from [42])

In the following, we provide an overview of the state-of-the-art related to test amplification including, test input data modification, test amplification for regression testing, and test case generation for behavioral models.

Test Input Data Modification. Data mutation testing [127] is a method inspired by the classical mutation testing for generating large test suites from a seed of a small set of test cases. The difference lies in how and where the mutation operators are applied. In mutation testing, the mutation operators are applied to the source code of a program to measure the adequacy of the test suite. Instead, data mutation applies mutation operators to the test input data for generating new test cases.

In the last years, this method has been applied for different purposes [135, 155, 157]. Sun et al. [135] propose a methodology for generating metamorphic relations. These relations are created by applying data mutation in the input relation. Then, a combination of constraint validation and generic mapping rules is used to generate output relations. Similarly, Zhu [157] introduces JFuzz, an automated framework for Java unit testing that combines data mutation and metamorphic testing for deriving and expressing metamorphic relations. Xuan et al. [155] present a proposal for detecting program failures by reproducing crashes through data mutation. In contrast to the previous approaches, their work does not focus on generating new test cases, but on updating the existing ones for triggering crashes on the program under study and therefore, finding errors.

Generating input test cases is also essential for fuzzy testing [156], which consists of generating random input data as a test case, and monitor the program for crashes or failing assertions. Fuzzers—the programs generating the inputs—can generate new inputs from scratch or modify existing ones using data mutation.

Test Amplification for Regression Testing. Several approaches use test amplification for regression testing. Xie [153] presents a framework for augmenting test suites with regression oracle checking. His proposal is supported by a tool, called Orstra, which focuses on asserting the behavior of JUnit test cases. For this purpose, Orstra amplifies automatically generated test suites by systematically adding assertions for improving their capability of avoiding regression faults. DSpot [43] targets the automatic amplification of JUnit test cases. It combines input space exploration [139] with regression oracle generation [153] techniques. The former is applied for putting the program under test in never explored states, and the latter aims at generating assertions for those new states. Given a set of manually-written JUnit test cases, DSpot generates variants of them which improve the mutation score. On the basis of DSpot, Abdi et al. [1] propose Small-Amp, an amplification approach for the Pharo Smalltalk ecosystem. Ebert et al. [126] provide a test amplification tool for Python. To this aim, the authors rely on the DSpot design, combining with Small-Amp features to alleviate the shortcomings related to dynamically typed languages.

Assis et al. [20] present an approach for test amplification of cross-platform applications. For this, the authors use four test patterns that analyze well-known features of a mobile application. The test input data is a sequence of events that is exchanged with the SUT and the input modifiers are defined using a set of test patterns specific to mobile applications.

Test Case Generation for Behavioral Models. Some researchers use MDE or other automated means to generate test cases from modeling artifacts, most notably from requirement models, use cases, or activity diagrams [11, 90, 122, 143]. Most of these efforts follow one of two main approaches for test case generation: path/coverage analysis [90, 122], or category partition [122, 143]. The former approach is based on analyzing all possible paths of behavior in the source model, and the latter partitions the requirements under test and generates test cases for combinations of such partitions. Differently from the objective of this thesis, these efforts are specific for models of system functional requirements, they do not assume an initial set of test cases, and do not propose any test case improvement technique.

Outside requirements modeling, test case generation for behavioral models has been handled using different methods. For example, Frolich and Link [58] generate test cases from Statecharts by translating the Statecharts into a planning problem, and using a planning tool to find test cases as solutions to the problem. Ahmadi and Hili [6] present an approach to automatically test components of UML-RT models with respect to a set of properties defined by state machines, and apply slicing to reduce the size of the components with respect to the properties. Rocha et al. [121] generate JUnit test cases from sequence diagrams via a transformation of the latter

into extended finite state machines. From fUML activity diagrams, Iqbal et al. [74] generate test cases with input data to cover all executable paths of the diagrams, together with their expected output. The interested reader can consult [5] for a recent survey on model-based testing using activity diagrams, including test case generation. In summary, test case generation for behavioral models has been tackled in the literature, but the proposals are normally language-specific. Moreover, these proposals do not target test amplification (i. e., improving an existing test suite).

Also in the modeling area, some research efforts focus on test case generation for model transformations, or transformation models. A test case in this scenario comprises an input model to the transformation and an oracle function. For example, Giron et al. [59] use software product lines and input metamodel coverage to generate a reduced set of test cases for transformations; Guerra and Soeken [66] use constraint solving to generate test models and partial oracles from declarative transformation specifications; Al-Azzoni and Iqbal [8] apply test case prioritization for regression of transformations based on an analysis of the transformation rules' coverage; and Troya et al. [141] infer likely metamorphic relations for ATL transformations, which can be used for metamorphic testing. The approach proposed by Troya et al. relies on the traces of the transformation executions to derive the metamorphic relations.

In summary, all existing test input data modification and test amplification approaches and tools target programs implemented by programming languages such as Java [43], Pharo Smalltalk [1], and Python [126]. However, this thesis aims at providing test amplification for executable models defined by xDSLs. Moreover, we find a variety of approaches for test case generation in the modeling area, but to our knowledge, test case improvement for xDSLs is not investigated yet. A potential test amplification approach for models can be a complement to these existing test case generation approaches to improve the quality of their generated test suites for regression testing.

2.4 Conclusion of the state-of-the-art

In this chapter, we presented all the material that are required to understand the contributions of this thesis as well as to situate our work within the related research work. In this section, we conclude this chapter by presenting a short and scoped systematic review of existing approaches that, as far as we know, aim to answer one or multiple of the challenges stated in the problem statement (discussed in Section 1.2 on page 2). These approaches were all initially introduced in Section 2.3.3 on page 28 and here we extract their features considering the objectives of this thesis. Table 2.2 presents our considered features and the possible alternatives for each of them, including:

- **Scope**: a testing framework can be either *DSL-Specific* or *generic*, meaning that it is either defined for a particular DSL or applies to a wide range of DSLs.
- **xDSL type**: the xDSLs supported by testing frameworks can be different regarding their execution semantics. They can be *compiled DSLs* having a translational semantics or *interpreted DSLs* having an operational semantics.
- **Testing level**: the definition of test cases can be supported in different levels including, the *model-level* i. e., defining test cases for the models conforming to the supported xDSL, or the *code-level* i. e., defining test cases for the code generated from the model or interpreting the model.
- **Test engineer**: the person who is enabled to define test cases can be a *domain expert* i. e., the user of the supported xDSL who defines the model under test and is not usually a programmer, or a *technical programmer* who has the expertise of a programming language (mainly when code-level testing is supported).
- **Test language**: the language used for defining test cases can be a *modeling language* that provides the domain concepts, or a *programming language* such as Java.
- **Test executability**: a testing framework supports the definition of executable test cases if it offers *test execution* on the models, *oracle validation* i. e., automatically examining whether the model under test behaved as expected by an executed test case, and *automatic reporting* i. e., producing a meaningful test execution result for the test engineer.
- **Advanced facilities**: a testing framework with advanced facilities supports *quality measurement* of test cases, *test debugging* of failed test cases, and *test amplification* to improve the written test cases.

Table 2.2 also presents the status of each related work regarding the mentioned features. Overall, among the 13 related works:

- ten are DSL-specific [18, 71, 74, 75, 87, 91, 97, 103, 106, 124] and three are generic to some extent [36, 105, 152].
- six are supporting compiled DSLs [71, 74, 75, 87, 124, 152], six interpreted DSLs [18, 91, 97, 103, 105, 106], and the work of Canizares et al. [36] targets both compiled and interpreted DSLs.
- nine allow to define test cases for the models [18, 36, 75, 87, 91, 97, 105, 106, 124] and the rest support code-level testing [71, 74, 103, 152].

Table 2.2: An overview of the state-of-the-art

Paper	Scope		xDSL type		Testing level		Test engineer		Test language		Test executability			Advanced facilities		
	DSL-Specific	Generic	Compiled DSLs	Interpreted DSLs	Model-level	Code-level	Domain expert	Technical programmer	Modeling language	Programming language	Test execution	Oracle validation	Automatic reporting	Quality measurement	Test debugging	Test amplification
Mens et al. [103]	•			•		•		•		•	•	•	•		•	
Iqbal et al. [75]	•		•		•		•		•	•	•	•	•	•	•	
Hili et al. [71]	•		•			•		•		•	•				•	
Santiago et al. [124]	•		•		•		•		•		•	•			•	
Kos et al. [87]	•		•		•		•		•		•	•	•		•	
Lubke and Van Lessen [97]	•			•	•		•		•		•	•	•			
Lazar et al. [91]	•			•	•		•		•		•	•	•			
Arnaud et al. [18]	•			•	•		•		•		•	•	•	•		•
Mijatov et al. [106]	•			•	•		•		•		•	•	•			
Iqbal et al. [74]	•		•			•		•		•	•	•	•	•		•
Wu et al. [152]		•	•			•		•		•	•	•	•		•	
Canizares et al. [36]		•	•	•	•		•		•		•					
Meyers et al. [105]		•		•	•		•		•		•	•				

- nine enable the domain expert to write test cases for models (the ones supporting model-level testing) [18, 36, 75, 87, 91, 97, 105, 106, 124] and the rest require a programmer to define test cases (the ones supporting code-level testing) [71, 74, 103, 152].
- nine use/define a modeling language for the definition of test cases [18, 36, 75, 87, 91, 97, 105, 106, 124] while five use a programming language [71, 74, 75, 103, 152] (the work of Iqbal et al. [75] uses both a modeling and a programming language).
- all the 13 approaches support test execution, 11 of them offer oracle validation [18, 74, 75, 87, 91, 97, 103, 105, 106, 124, 152], and nine of them provide automatics reporting [18, 74, 75, 87, 91, 97, 103, 106, 152].
- quality measurement is mentioned in three related work [18, 74, 75], test debugging is supported by six approaches [71, 75, 87, 103, 124, 152] (four of them indeed rely on the debugging facilities of their considered programming languages [71, 75, 103, 152]), and two approaches consider improving test cases but not necessarily with test amplification technique [18, 74].

In this thesis, we aim at proposing a *generic* testing framework for *intereprted DSLs* which allows the *domain experts* to define *executable* test cases for *models* and

provides *advanced testing facilities* for them. To the best of our knowledge, none of the existing approaches fulfill simultaneously all the requirements we considered.

Chapter 3

Test Case Definition and Execution

3.1 Introduction

A testing framework must at least include (*i*) a way to write test cases, and (*ii*) a way to execute such test cases in unison with the programs or models under test. Providing these testing facilities for a given *new* xDSL remains an expensive and error-prone task due to three interconnected challenges. First, to allow the domain expert to write test cases for models, a testing language must be defined, generated, or identified. In particular, this testing language must somehow allow the domain expert to use domain concepts to define how a model under test should be executed, and what results should be expected from the execution. Second, the execution semantics of this testing language must somehow be connected to the execution semantics of the considered xDSL, for the testing language to demand the execution of models as needed. Third, this testing language must provide facilities to analyze the runtime behavior of the tested model and to compare it with the expected one.

A recent effort of the European Telecommunications Standards Institute (ETSI) led to the creation of the Test Description Language (TDL) [98]. Since TDL is not specific to any specific GPL or xDSL, it represents an interesting candidate for generically writing test cases for executable models. In addition, TDL was designed as a simple language for testers lacking programming knowledge, making it a good fit for domain experts working on models. Unfortunately, TDL fails to fully address the three aforementioned challenges:

- because of its genericity, TDL requires the domain expert to first define the required domain-specific concepts, before being able to write test cases.

- the TDL standard does not provide any clear way to make TDL test cases able to execute models conforming to a given xDSL.

- the TDL standard relies on a simple representation of the expected observable behavior of the system under test and does not provide any efficient way to analyze an arbitrarily complex runtime behavior of a tested model.

In a context where the engineering of new xDSLs is recurrent, a desirable solution would be generic test case definition and execution facilities that can be systematically applied to any given xDSL, indeed be used for testing the models conforming to the xDSL. In this section, we address the above-mentioned limitations and thereby propose such generic testing facilities for xDSLs. We use TDL as a testing language by relying on three main contributions.

First, we provide a model transformation to automatically generate a TDL library—i. e., all the TDL boilerplate code that the domain expert would otherwise write by hand—from the definition of an xDSL. Such generated TDL library can be used by the domain expert to write test cases for models (conforming to the considered xDSL) using the domain concepts.

Second, we provide an operational semantics for TDL, adapted to the testing of executable models. To be compatible with a wide range of diverse xDSLs, this operational semantics for TDL is not coupled to any specific xDSL, nor to any specific metaprogramming approach used to define the considered xDSL.

Third, the approach provides three different methods to interrogate the runtime behavior of the tested model: (*i*) relying on model comparison; (*ii*) relying on the xDSL's behavioral interface as it defines the observable events that a running model may expose; and (*iii*) relying on an Object Constraint Language (OCL) interpreter. This enables the definition of oracles for executable models in TDL test cases.

In the following, we first present an overview of our proposed testing facilities for xDSLs (Section 3.2). We then describe what should a TDL test case look like through examples in Section 3.3. Afterward, each provided facility is individually detailed in Sections 3.4, 3.5, and 3.6. Section 3.7 presents our tool support and Section 3.8 shows an empirical evaluation of the provided facilities. Finally, Section 3.9 concludes this chapter with an outline for future work.

3.2 Overview

Figure 3.1 presents an overview of our proposed testing facilities for xDSLs[1], supporting the definition and the execution of TDL test cases for the conforming

[1]Elements of the Figure are written in italic in the text.

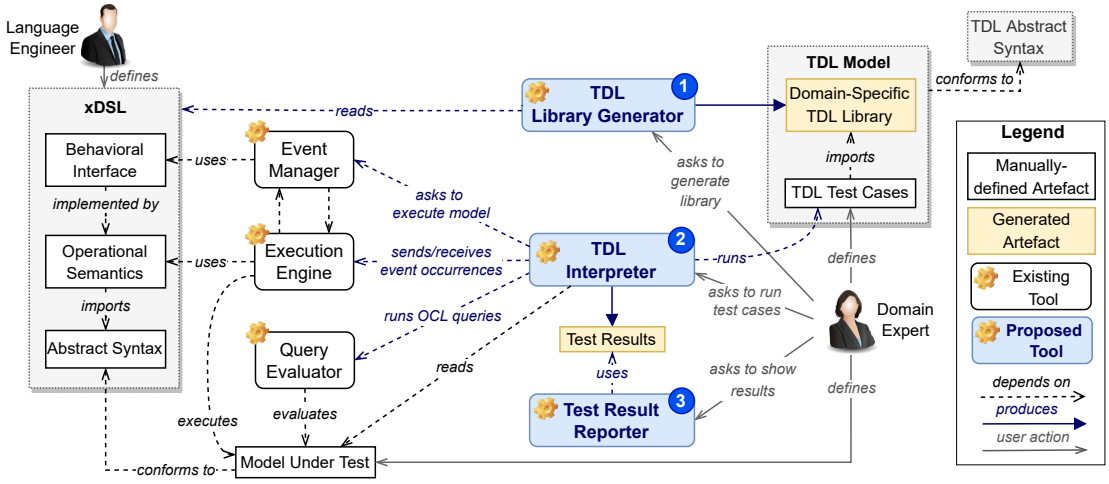


Figure 3.1: An overview of the proposed test case definition and execution facilities

executable models. At the top left corner, we assume that an xDSL either non-reactive or reactive was implemented by a language engineer according to the definitions given in Section 2.2. On the right, the domain expert uses the provided xDSL to define an executable model and wishes to write TDL test cases for this model to ensure it behaves as expected.

The *TDL Library Generator* (label 1) automatically generates a domain-specific TDL Library that the domain expert can then use to conveniently write test cases for executable models conforming to the given xDSL. This library provides all the data types required for the specification of test data, a set of default test configurations, elements for requesting the execution of the models under test, and elements for writing OCL queries [41] in the TDL test cases. Moreover, when the given xDSL is reactive, hence having a behavioral interface (at the top left corner), the generated library also provides an event-compatible TDL package comprising the required elements for writing and executing event-driven TDL test cases for reactive models. As shown in Figure 3.1, the library generator requires as input data the definition of the xDSL. In particular, the abstract syntax, the part of the operational semantics defining the possible runtime states of the conforming models, and the behavioral interface (for reactive xDSLs). More details are provided in Sections 3.4.

Executing TDL test cases on the models is the role of the *TDL Interpreter* component (label 2). This interpreter is based on an operational semantics for TDL, adapted to the testing of executable models. For this purpose, it is connected to three external components: the *Execution Engine*, the *Query Evaluator*, and the *Event Manager*. We assume that the *Execution Engine* exists and provides services to trigger the execution of a model conforming to an xDSL. In particular, that this

engine is able to load a model, load an xDSL, and execute the model using the operational semantics of the xDSL. Here, the execution engine is used by the TDL Interpreter to start the execution of a model, to get the content of the model, or to set the model in a specific runtime state.

We also consider that the *Query Evaluator* can evaluate OCL queries [41]. This component is used by the TDL Interpreter to evaluate queries written inside TDL test cases, so the query validation result can be used as part of the oracle of a TDL test case. Finally, we assume that an *Event Manager* exists and provides services to interact with a running reactive model. More precisely, given a reactive xDSL, it enables the external tools such as testing tools to exchange events conforming to the xDSL's behavioral interface with the models conforming to the xDSL's abstract syntax at runtime. We provide a detailed explanation of the *TDL Interpreter* in Section 3.5.

At the end of the test execution, the TDL Interpreter produces a report of test results. The *Test Result Reporter* component (label 3) visualizes this report in a user interface as well as serializes it as a persistent file. Accordingly, the domain expert can use this component to investigate the test execution results. In Section 3.6, more details are given.

With everything in place, the domain expert can use the generated domain-specific TDL library to write test cases, can then use the TDL interpreter to execute them, and can see the results and save them. In subsequent, we describe what should a TDL test case look like through examples. Then we introduce each proposed component in detail.

3.3 Samples of TDL Test Cases

Depending on how the operational semantics of an xDSL is defined (i. e., content-based when the xDSL is non-reactive, event-driven when the xDSL is reactive), its conforming models will be executed differently. As already mentioned in Section 2.2.3, the execution of a non-reactive model is performed in a *closed* environment while a reactive model can interact with the external environment during its execution (based on the xDSL's behavioral interface). This difference has a direct impact on how we write and execute test cases on models. To make this difference clearer, we wrote one test case for the xArduino sample model in two styles, as presented below.

3.3.1 A Sample Test Case for a Non-Reactive Model

Closed execution of non-reactive models means (a) it is possible to provide an initial runtime state for the model before its execution is started; and (b) it is

possible to retrieve the final runtime state of the model at the end of its execution. Therefore, in a prospective test case of a non-reactive model, the test input data and the expected output are both the model under test in different runtime states. For example, if we define such a test case for the xArduino model of Figure 2.2 on page 12, it would look like Figure 3.2(**a**). It is defined as a scenario of exchanging messages between a Test Component and the System-Under Test (SUT), in this case, the xArduino model. This test case checks whether the LED turns on when the button is pressed and whether the alarm alternates between noise/silence periods when the sensor detects an obstacle.

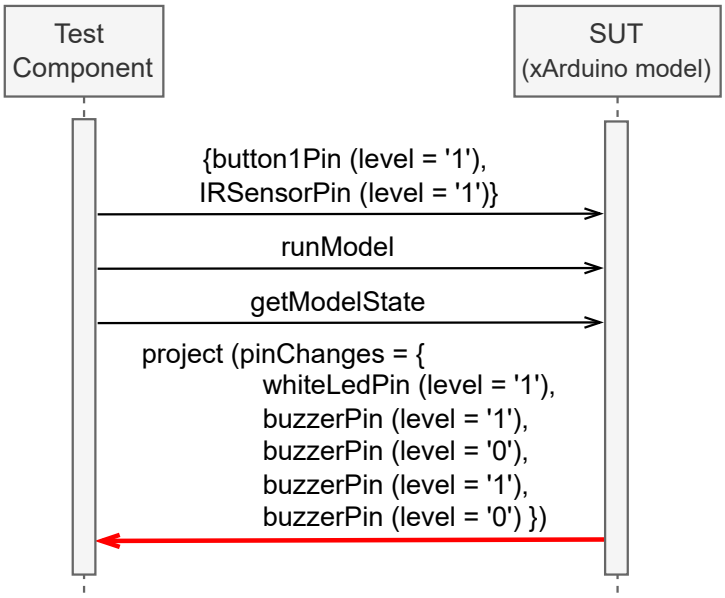
First, the test component sets the model in a specific runtime state where the `button1` and the `infrared sensor` are turned on. Then it requests for running the model under test and retrieving its runtime state after the execution. The test case expects to observe that the `white Led` turns on and the `buzzer` turns on and off two times. This expected output is defined as a list of changes of the level of `whiteLedPin` and `buzzerPin`, captured as the value of pinChanges dynamic feature (i. e., part of the runtime state definition already shown in Figure 2.3 on page 15). The assertion fails because due to the defect of the model (highlighted in red in Figure 2.2 on page 12), the `buzzer` turns on and off only one time.

3.3.2 A Sample Test Case for a Reactive Model

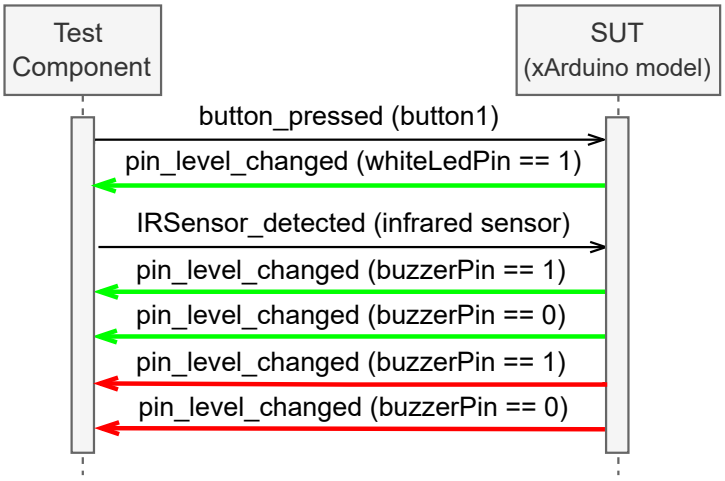
As the behavioral interface of a reactive xDSL defines how to interact with the conforming models, a prospective test case for a reactive model should be described as a scenario in which the test system sends instances of accepted events to the model and checks whether the model sends back the expected exposed events instances. For example, Figure 3.2(**b**) shows such a test case for the xArduino model of Figure 2.2 on page 12, considering its reactive execution.

The test data are instances of the events specified by the behavioral interface of the xArduino DSL (Figure 2.5(**c**) on page 19). The values of the events' parameters are elements of the xArduino model with values for their runtime features, such as `buzzerPin` element with value 1 for its `level` feature. This test case has the same logic as the test case of Figure 3.2(**a**). Due to the defect of the xArduino model (highlighted in red in Figure 2.2 on page 12), the first three assertions pass but the last two fail; their corresponding arrows in Figure 3.2(**b**) are highlighted in green and red, respectively.

Therefore, depending on how the operational semantics of an xDSL is defined (i. e., content-based when the xDSL is non-reactive, event-driven when the xDSL is reactive), test cases for its conforming models must be defined and executed differently. In the remainder of the section, we explain how our proposed approach provides facilities for the domain expert to write such test cases for any executable model.



(a) The TDL test case when executed by the xArduino *content-based* semantics. It has one failed assertion (highlighted in red).



(b) The TDL test case when executed by the xArduino *event-driven* semantics. It has two passed and two failed assertions (highlighted in green and red, respectively).

Figure 3.2: A potential TDL test case for the xArduino model of Figure 2.2 on page 12 that is written in two styles to be executed by two different xArduino execution semantics.

3.4 TDL Library Generator

This section presents the *TDL Library Generator*, whose role is to produce a TDL library specific to a given xDSL. We first present how the component works and then explain what are the contents of the generated TDL library. Afterward, we show how such a library can be used by the tester (i. e., the domain expert) to write test cases for models conforming to the xDSL.

3.4.1 Description of the Library Generator

Figure 3.3 illustrates a detailed overview of the TDL Library Generator. As can be seen, the generator reads the definition of an xDSL and produces a TDL library specific to the xDSL providing a set of TDL elements for the tester. More specifically, the TDL library generated for each given xDSL contains a set of TDL Packages:

1. **xDSL-Specific Types Package**, containing the TDL data types required for the specification of test data. They are generated by a model transformation from the Ecore metamodel of the xDSL (i. e., its abstract syntax and the parts of its operational semantics defining the possible runtime states of the conforming models) to TDL.

2. **xDSL-Specific Events Package**, having the TDL definition of the events of the xDSL's behavioral interface. They are needed for the specification of test data when testing reactive models. This package is generated through a model transformation from behavioral interface of the given xDSL to TDL.

3. **Common Package**, providing TDL elements common to any given xDSL, including a set of elements for performing operations on the model under test and elements for enabling the use of OCL queries in the test cases.

4. **Test Configuration Package**, providing a default test configuration to be used by the TDL test cases written for executable models.

The *xDSL-Specific Events Package* is generated solely for reactive xDSLs while the others are generated for any xDSL (i. e., either non-reactive or reactive). As can be seen in Figure 3.3, the *Common Package* and the *Test Configuration Package* are provided by a code generator (it only requires the name of the xDSL as input).

In what follows, we present in order how each package is generated. At the end, we show how they can be used for writing test cases for executable models.

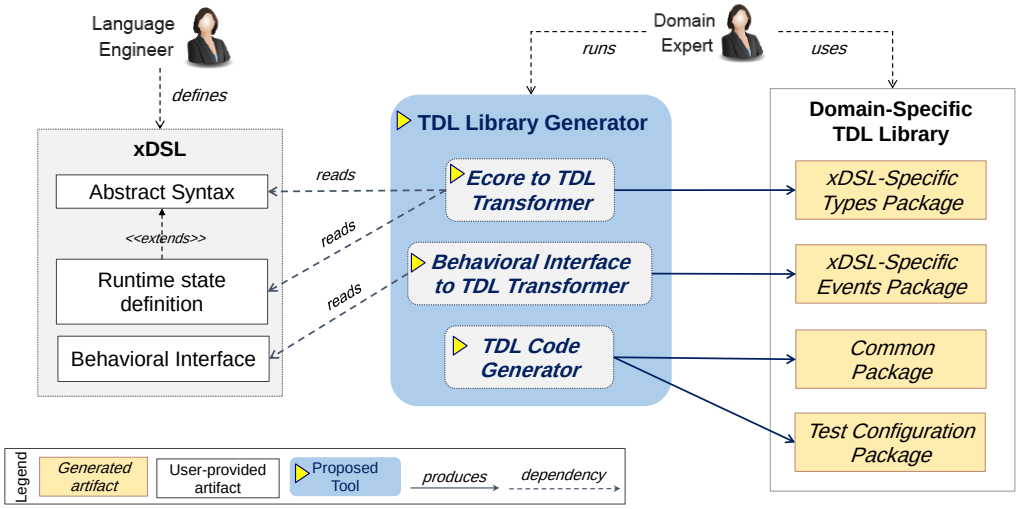


Figure 3.3: Detailed overview of the TDL library generator

3.4.2 Generation of the xDSL-Specific Types Package

As discussed in Section 2.3.2 on page 25, to use TDL for a specific domain, all data types required for the specification of test data have to be defined beforehand. Yet, in the context of testing executable models, we can observe that these data types are in fact already defined as part of the definition of the considered xDSL, both in the abstract syntax and in the definition of the possible runtime states in the operational semantics. It is therefore possible to infer the required TDL data types from this information.

As explained in Section 2.2 on page 10, we consider that the Ecore metamodel of the xDSL includes the abstract syntax and is extended by merging the definition of the possible runtime states of conforming models. Accordingly, we rely on a model transformation from Ecore to TDL to automatically generate the required TDL data types for testing executable models conforming to a given xDSL.

A summary of the transformation rules is shown in Table 3.1. Each rule takes one element from the left column and transforms it into an element of the same row of the right column. In a nutshell, the objective of this transformation is to transform each Ecore EClass into a TDL data type, either simple or structured. An EClass containing EStructuralFeatures (i. e., attributes and references) is transformed to a StructuredDataType containing Members, each of which corresponds to one feature of the EClass. To distinguish abstract classes from concrete ones and dynamic features from static ones, annotations are generated and assigned to the corresponding element. An inheritance relationship between two classes is transformed into an Extension relationship in TDL. The transformation generates

Table 3.1: Outline of the Ecore to TDL transformation rules

Source Ecore element	**Target TDL element**
`EClass` with no `EStructuralFeature`	`SimpleDataType`
`EClass` containing `EStructuralFeature`	`StructuredDataType` containing one `Member` per `EStructuralFeature`
Abstract `EClass`	an `Annotation` of 'abstract' `AnnotationType` is set to its corresponding `DataType`
Inherited `EClass`	an `Extension` of the `DataType` generated for each of its super classes, assigns to its related `DataType`
`EStructuralFeature` (`EAttribute` and `EReference`)	`Member` contained in the `StructuredDataType` that is generated for its container `EClass`. Its type is the `DataType` corresponding to the feature `eType`
Dynamic `EStructuralFeature` (having a 'dynamic' `EAnnotation`)	an `Annotation` of 'dynamic' `AnnotationType` is set to its corresponding `Member`
`EDataType`	`SimpleDataType`
`EEnum`	`SimpleDataType`
`EEnumLiteral`	`SimpleDataInstance` that its type is the `SimpleDataType` of the related `EEnum`
`EPackage`	`Package` containing all the generated elements as `packagedElement`

simple TDL data types for Ecore primitive data types (EDataType) and Enums, and EnumLiterals are transformed to the instances of the TDL data type corresponding to their related enum. Finally, an Ecore EPackage is transformed to a TDL Package that is the root container of all the generated TDL elements.

We used the ATL language [82] to define the transformation rules. An excerpt of this transformation is shown in Listing 3.1 on page 56, with the ATL code for transforming a concrete Ecore class containing features and superclasses, to a TDL structured data type containing members and an extension for each of its super classes. This rule has several calls to other transformation rules that are provided in Appendix A.

Listing 3.2 on page 57 presents the TDL DataTypes generated from the Ecore metamodel of the non-reactive xArduino (Parts (**a**) and (**b.1**) of Figure 2.6 on page 22). The `abstract` Annotation element is for specifying the TDL DataTypes generated for an abstract EClass, such as `Project` (line 6) and `Board` (line 7).

```
1  rule concreteInheritedClass2structuredType {
2      from class: Ecore!EClass (class.eAllStructuralFeatures.notEmpty()
3                                and class.eSuperTypes.notEmpty()
4                                and not class.abstract)
5      to type: TDL!StructuredDataType(
6          name ← class.name,
7          member ← class.eStructuralFeatures→ collect(f |
8                  if (f.isDynamicFeature)
9                      then thisModule.dynamicFeature2annotatedMember(f)
10                     else thisModule.staticFeature2member(f)
11                     endif),
12         extension ← class.eSuperTypes → collect (st | thisModule.
       superClass2extension(st)))
13 }
```

Listing 3.1: Example of an Ecore to TDL transformation rule

Moreover, to distinguish the TDL DataTypes generated for the dynamic elements, an Annotation named `dynamic` is defined and assigned to them, such as the `pinChanges` member of the Project (line 5) and the `level` member of the DigitalPin (line 10).

The only difference between the Ecore metamodel of the non-reactive and reactive xArduino is the `pinChanges` dynamic feature i. e., defined only for non-reactive xArduino (refer to Figure 2.6 on page 22). Consequently, the xDSL-specific types package for the reactive xArduino (generated from parts (**a**) and (**c.1**) of Figure 2.6 on page 22) would be the same as listing 3.2 but without the `pinChanges` Member, as shown in listing 3.3 on page 57. Using these generated packages, the domain expert can easily define model elements along with their runtime states in TDL and use them as test data.

3.4.3 Generation of the xDSL-Specific Events Package

As discussed in Section 3.3.2 on page 51, to write test cases for a model conforming to a reactive xDSL, we need to use the events of the xDSL's behavioral interface as test data types. This means the testing language should support using the events for the definition of test data. Since our approach uses the TDL testing language, we need the definition of the events in TDL, which we provide by the *xDSL-Specific Events* package.

This package is automatically generated by a transformation from the behavioral interface metamodel (already shown in Figure 2.4 on page 18) to TDL. Table 3.2 shows the outline of the transformation rules. In a nutshell, a `BehavioralInterface` is transformed to a TDL `Package` that is the container of other elements.

```
1  Package xArduinoTypes_nr {
2      Type Project (
3          boards of type Board,
4          sketches of type Sketch,
5          pinChanges of type DigitalPin with {dynamic;}
6      ) with {abstract;};
7      Type Board (pins of type DigitalPin) with {abstract;};
8      Type Sketch (block of type Block);
9      Type DigitalPin (
10         level of type EInt with {dynamic;},
11         module of type Module
12     ) with {abstract;};
13     Type Module (
14         _name of type EString
15     ) with {abstract;};
16     Type PushButton extends Module();
17     Type InfraRedSensor extends Module();
18     ...
19 }
```

Listing 3.2: Some of the TDL data types generated for the non-reactive xArduino

```
1  Package xArduinoTypes_r {
2      Type Project (
3          boards of type Board,
4          sketches of type Sketch
5      ) with {abstract;};
6      ...
7  }
```

Listing 3.3: Some of the TDL data types generated for the reactive xArduino

Each `Event` is transformed to a `StructuredDataType` which is annotated according to the `EventType` and comprises `Members` generated for the `EventParameters`. To assign the type of `Members`, the content of the previously generated *xDSL-Specific Types Package* is used by creating an `Import` element.

Listing 3.4 shows the *xDSL-Specific Events Package* generated for the behavioral interface of the reactive xArduino (Figure 2.6(**c.3**) on page 22). To distinguish *accepted events* from *exposed events*, two Annotation elements are generated (lines 4-5). For each event of the behavioral interfaces, a TDL Type is produced (lines 7-13). Each event is annotated with one of the Annotation elements according to the type of the event.

The parameters of the events are transformed to Members of the TDL Types. For example, line 7 shows the Member generated for the *button* parameter of the

Source BI element	Target TDL element
`Behavioral-Interface`	`Package` containing all the other generated elements
-	`Import` the xDSL-Specific Types `Package`
`EventType`	`AnnotationType`
`Event`	`StructuredDataType` containing one `Member` per event parameter and an `Annotation` based on it type
`EventParameter`	`Member`. Its type is set using the TDL `DataTypes` provided by the imported `Package`

Table 3.2: Behavioral interface to TDL transformation rules

```
1  Package xArduinoEvents {
2      Import all from xArduinoTypes_r;
3
4      Annotation AcceptedEvent;
5      Annotation ExposedEvent;
6
7      Type button_pressed (button of type PushButton)
8          with {AcceptedEvent;};
9      Type button_released (button of type PushButton)
10         with {AcceptedEvent;};
11     Type IRSensor_detected (sensor of type InfraRedSensor)
12         with {AcceptedEvent;};
13     Type pin_Level_Changed (pin of type DigitalPin)
14         with {ExposedEvent;};
15 }
```

Listing 3.4: TDL elements generated for the xArduino behavioral interface

button_pressed event. Since the parameters are references to the model elements, their type conforms to the xDSL's abstract syntax. Thanks to the generated *xDSL-Specific types package*, we have the definition of all the required data types in TDL. Therefore, we can use them to assign the type of the Members. To this end, this package is imported to the *xDSL-Specific events package* (line 2) and its content i. e., the TDL Types generated for the xArduino metamodel is used several times (e. g., `PushButton` in line 7 or `InfraRedSensor` in line 11).

3.4.4 Generation of the Common Package

This package contains common elements that are not specific to the given xDSL, but provide common testing facilities for any xDSL. As shown in Listing 3.5, the `Verdict Type` (line 2) along with several instantiations of it (lines 3-5) are

defined to be used for test verdict assignment. This Package also provides elements for performing several operations on the model under test (later referred to as *model execution commands*), including `runModel` for executing the model (line 8), `resetModel` for resetting its state to the default (line 9), and `getModelState` for getting its current state, i. e., the content of its dynamic features (line 10). To enable the tester to use OCL queries in the test cases, this package provides a data type named `OCL` (line 12) and an instantiation of it (line 13). This instantiation uses the question mark TDL symbol (`?`) for the `context` and `query` attributes, meaning that a value must be given to them when the `oclQuery` instance is used.

```
1  Package common {
2      Type Verdict;
3      Verdict PASS;
4      Verdict FAIL;
5      Verdict INCONCLUSIVE;
6
7      Type modelExecutionCommand;
8      modelExecutionCommand runModel;
9      modelExecutionCommand resetModel;
10     modelExecutionCommand getModelState;
11
12     Type OCL ( context of type EObject , query of type EString );
13     OCL oclQuery ( context = ? , query = ? );
14 }
```

Listing 3.5: TDL common package for all xDSLs

3.4.5 Generation of the Test Configuration Package

In TDL, a test case must refer to a *test configuration* defining what is the system under test, and how to communicate with it. In particular, a test configuration can define what are the available communication *gates*, each gate allowing specific types of messages. In the present approach, we consider that three kinds of messages can be exchanged with the model under test:

1. *model execution commands* related to the execution of the model and getting access to its runtime state.
2. *OCL commands* related to the execution of the OCL queries.
3. *Event-based commands* related to the exchange of the event occurrences with a running reactive model.

Accordingly, given an xDSL, our generator will generate a TDL Test Configuration Package introducing these gates and components for the xDSL, along with a

```
1  Package testConfiguration_nr {
2      Import all from common;
3
4      Gate Type genericGateType accepts modelExecutionCommand;
5      Gate Type oclGateType accepts OCL;
6      Component Type component having {
7          gate genericGate of type genericGateType;
8          gate oclGate of type oclGateType;
9      }
10     Annotation MUTPath;
11     Annotation DSLName;
12
13     Test Configuration xArduinoConfiguration_nr {
14         create Tester tester of type TestSystem;
15         create SUT arduino of type MUT with {
16             MUTPath : 'TODO : Put the path to the MUT';
17             DSLName : 'non-reactiveArduino';
18         };
19         connect tester.genericGate to arduino.genericGate;
20         connect tester.oclGate to arduino.oclGate;
21   }
22 }
```

Listing 3.6: TDL test configuration package generated for the non-reactive xArduino

test configuration that makes use of them. Listing 3.6 shows the *Test Configuration Package* generated for the **non-reactive** xArduino. It has two Gate Types, including `genericGateType` for exchanging `modelExecutionCommands` (line 4) and `oclGateType` for exchanging `OCL` queries (line 5) which are provided by the `common` Package (imported in line 2). There is a Component Type comprising one gate instance for each Gate Type (lines 6-9).

Finally, a Test Configuration is defined containing two Component Instances, one of the Tester kind (line 14) and one of the SUT kind (lines 15-18). The SUT requires information about the model under test, including the path to the model under test (the `MUTPath` annotation in line 16) that should be set by the domain expert, and the name of the DSL that the model conforms to (the `DSLName` annotation in line 17) which is automatically set by the *TDL Library Generator* based on the given xDSL, here the non-reactive xArduino. The test configuration also specifies how the test system connects to the SUT through the definition of the Connections between their Gate instances (lines 19-20).

Listing 3.7 presents the *Test Configuration Package* generated for the **reactive** xArduino. Similarly to the one generated for the non-reactive xArduino (Listing 3.6 on page 60), it has the two Gate Types for exchanging model execution commands (line 5) and OCL queries (line 6). In addition, it has a specific Gate Type for

```
1  Package testConfiguration_r {
2      Import all from common;
3      Import all from xArduinoEvents;
4
5      Gate Type genericGateType accepts modelExecutionCommand;
6      Gate Type oclGateType accepts OCL;
7      Gate Type reactiveGateType accepts button_pressed , button_released ,
       IRSensor_detected ,pin_level_changed ;
8      Component Type component having {
9          gate genericGate of type genericGateType;
10         gate oclGate of type oclGateType;
11         gate reactiveGate of type reactiveGateType;
12     }
13     Annotation MUTPath;
14     Annotation DSLName;
15
16     Test Configuration xArduinoConfiguration_r {
17         create Tester tester of type component;
18         create SUT arduino of type component with {
19             MUTPath: 'TODO : Put the path to the MUT';
20             DSLName: 'reactiveArduino';
21         };
22         connect tester.genericGate to arduino.genericGate;
23         connect tester.oclGate to arduino.oclGate;
24         connect tester.reactiveGate to arduino.reactiveGate;
25     }
26 }
```

Listing 3.7: TDL test configuration package generated for the reactive xArduino

communicating events i. e., `reactiveGateType` (line 11). The TDL definition of the events is provided by the `xArduinoEvents` Package (imported in line 3). Accordingly, the definition of the Component Type (lines 8-12) is extended with a new instance of the `reactiveGateType` (line 11). The definition of the Test Configuration element (line 16) is similar to the one generated for the non-reactive xArduino (line 13 of listing 3.6 on page 60) but for two differences: the name of the DSL is changed (the `DSLName` annotation in line 20) and a new connection between reactive gates is added (line 24).

3.4.6 Using the TDL Library to Write Test Cases

In Section 3.3, we described an overview of two test cases for the xArduino sample model (Figure 2.2 on page 12): (i) Figure 3.2(**a**) for its non-reactive execution; and (ii) Figure 3.2(**b**) for its reactive execution. By using the TDL Library generated for the xArduino, the domain expert can write such test cases in TDL that will be

executable. They are shown in (i) lines 14-29 of listing 3.8; and (ii) lines 14-29 of listing 3.9, respectively.

Using the generated data types (the `xArduinoTypes_nr` package imported in line 3 of listing 3.8, and the `xArduinoTypes_r` package imported in line 3 of listing 3.9), the domain expert can define model elements to use them as test data, such as using DigitalPin data type to define different pins of the xArduino model (lines 6-11 of listing 3.8) or using InfraRedSensor data type to define the `infrared sensor` element (lines 8-11 of listing 3.9).

The test case for the non-reactive xArduino uses the `xArduinoConfiguratio_nr` (line 14) provided by the `testConfiguration_nr` package i. e., imported in line 4. Likewise, the test case for the reactive xArduino uses the `xArduinoConfiguratio_r` (line 14) provided by the `testConfiguration_r` package i. e., imported in line 5.

Both test cases are defined as a sequence of exchanging data and/or requests between the gates of the `Tester` and `SUT` component instances. When test data is a *model execution command* or a runtime state, it should be exchanged through the `genericGate` of the components (such as lines 15-24). When the data is either an OCL query or an expected output related to the query evaluation result, the `oclGate` should be used, such as lines 25-28 of listing 3.8 where the tester queries the level of the buzzer pin and expects it to be '0'.

In addition, when test data is an event instance (i. e., in the test cases of reactive models), they must be exchanged through the `reactiveGate` of the components. For example, in listing 3.9, the `xArduinoEvents` package is imported (line 4) which allows using the event instances as test data (lines 15-17). In line 15, the `tester` sends a `button_pressed` event for the `button1` to the model under test, so the event is used as test input data. Afterward, an assertion is defined where the expected output is a `pin_Level_Changed` event for the `whiteLedpin` (line 17), so the event is used as expected output; according to the TDL semantics, when the sender of a TDL Message is the component instance with the SUT role, the Message is an assertion and its carried data is the expected output.

So far, we have presented the result of using the proposed facilities for providing testing support for the xArduino DSL. To better demonstrate the genericity of the approach, another example is provided in Appendix B for a different xDSL named xPSSM that is a reactive xDSL for describing UML state machines.

3.5 TDL Operational Semantics for xDSLs

In this section, we present an operational semantics for TDL tailored for the testing of executable models. We initially present how we refined the execution semantics provided in the TDL standard for the testing of executable models, We then define

```
Package TestSuite4nonReactive {
    Import all from common;
    Import all from xArduinoTypes_nr;
    Import all from testConfiguration_nr;

    //test data
    Project project (pinChanges = ?);
    DigitalPin IRSensorPin(_name = "IRSensorPin");
    DigitalPin whiteLedPin (_name = "whiteLedPin", level =?);
    DigitalPin button1Pin (_name = "button1Pin");
    DigitalPin buzzerPin (_name = "buzzerPin", level =?);

    //test cases
    Test Description test1 uses configuration xArduinoConfiguration_nr{
        tester.genericGate sends {
            button1Pin (level = '1'), IRSensorPin (level = '1')}
            to arduino.genericGate;
        tester.genericGate sends runModel to arduino.genericGate;
        tester.genericGate sends getModelState to arduino.genericGate;
        arduino.genericGate sends project (pinChanges = {
                whiteLedPin (level = '1'),
                buzzerPin (level = '1'), buzzerPin (level = '0'),
                buzzerPin (level = '1'), buzzerPin (level = '0')
            }) to tester.genericGate;
        tester.oclGate sends oclQuery
            (context = buzzerPin, query = "self.level")
            to arduino.oclGate;
        arduino.oclGate sends '0' to tester.oclGate;
    }
}
```

Listing 3.8: A TDL test case for the non-reactive execution of the running example

the operational semantics itself and explain how it is decoupled from both the xDSLs and the metaprogramming approaches used for their implementation.

3.5.1 Adapting TDL Semantics to Model Execution

The TDL Standard consists of several documents: the TDL metamodel, the TDL graphical syntax, the TDL exchange format, the UML profile for TDL, the mapping from TDL to TTCN-3, among others[2].

Two parts of the standard are related to the semantics of TDL. First, the *metamodel* document [52] specifies the abstract syntax as a metamodel, and its associated semantics using natural language. It introduces the basic principles

[2] https://tdl.etsi.org/index.php/downloads

```
Package TestSuite4reactive {
    Import all from common;
    Import all from xArduinoTypes_r;
    Import all from xArduinoEvents;
    Import all from testConfiguration_r;

    //test data
    InfraRedSensor IRSensor(_name = "infrared sensor");
    DigitalPin whiteLedPin (_name = "whiteLedPin", level =?);
    PushButton button1 (_name = "button1");
    DigitalPin buzzerPin (_name = "buzzerPin", level =?);

    //test cases
    Test Description test1 uses configuration xArduinoConfiguration_r{
        tester.reactiveGate sends button_pressed (
            button = button1) to arduino.reactiveGate;
        arduino.reactiveGate sends pin_Level_Changed (
            pin = whiteLedPin (level = '1')) to tester.reactiveGate;
        tester.reactiveGate sends IRSensor_detected (
            sensor = IRSensor) to arduino.reactiveGate;
        arduino.reactiveGate sends pin_Level_Changed (
            pin = buzzer_pin (level = '1')) to tester.reactiveGate;
        arduino.reactiveGate sends pin_Level_Changed (
            pin = buzzer_pin (level = '0')) to tester.reactiveGate;
        arduino.reactiveGate sends pin_Level_Changed (
            pin = buzzer_pin (level = '1')) to tester.reactiveGate;
        arduino.reactiveGate sends pin_Level_Changed (
            pin = buzzer_pin (level = '0')) to tester.reactiveGate;
    }
}
```

Listing 3.9: An event-driven TDL test case for the reactive execution of the running example

of TDL and describes all the test-specific concepts included in the TDL abstract syntax, categorized as Foundation, Data, Time, Test Configuration, and Test Behavior. For each concept, the semantics, the relationship with other concepts, the properties, and the static constraints are also provided. Second, the *mapping to TTCN-3* document is essentially a translational semantics for TDL using TTCN-3 as a target language. While this semantics does allow the execution of TDL test cases, it is only partial and mainly aims to manage test cases for software systems communicating through common protocols (TCP, UDP, SQL, HTTP, etc.). Therefore, we deemed this translational semantics too distant from the aim of this work—i. e., the testing of executable models. Instead, we solely used the execution semantics described in the *metamodel* document as the reference specification for

the operational semantics we propose.

However, as previously explained, our approach aims to cover two additional concerns: managing the execution of models (both non-reactive and reactive), and managing OCL queries. As these concerns are not covered by the standardized TDL semantics, we have to make adaptations for the operational semantics we propose. In Section 3.4, the *TDL Library Generator*, takes these concerns into account by generating specific *data types* (`ModelExecutionCommand` and `OCL` in the *Common Package* and the content of the *xDSL-Specific Events Package*) along with *gate types* accepting data conforming to them (`genericGateType`, `oclGateType`, and `reactiveGateType` in the *Test Configuration Package*). Accordingly, our operational semantics must be able to interpret such generated elements which is presented in Section 3.5.3.

In addition, to specify the result of a test case execution, we support the verdicts provided by the TDL *metamodel* document, including PASS, FAIL, and INCONCLUSIVE [52]. PASS and FAIL correspond to observing valid and invalid behaviors of the SUT, respectively, while INCONCLUSIVE is used when neither pass nor fail can be assigned. For instance, if the *tester* sends a syntactically wrong OCL query to the *SUT*, the TDL Interpreter will interrupt the test case execution and the verdict will be assigned as INCONCLUSIVE.

It should be noted that at the moment, we do not support all the TDL elements, such as complex Behavior concepts (e. g., Parallel, Exceptional, Periodic). These concepts enable the tester to define different types of tests such as load tests and distributed tests, so we consider them as future work.

3.5.2 Required External Components

As illustrated in Figure 3.1, the TDL Interpreter needs connections with three external components. We assume they already exist and provide services as follows:

- **Execution Engine**: provides services to manage the execution of the models such as running the model, resetting its state to default, and getting its current state. This component uses the operational semantics of an xDSL to execute its conforming models.
- **Query Evaluator**: can trigger the evaluation of an OCL query on a model and retrieves the result.
- **Event Manager**: provides services to send event occurrences to a running reactive model and to receive event occurrences exposed by the model. As running the model is performed by an execution engine, this component is also connected to the execution engine to communicate event occurrences with running models.

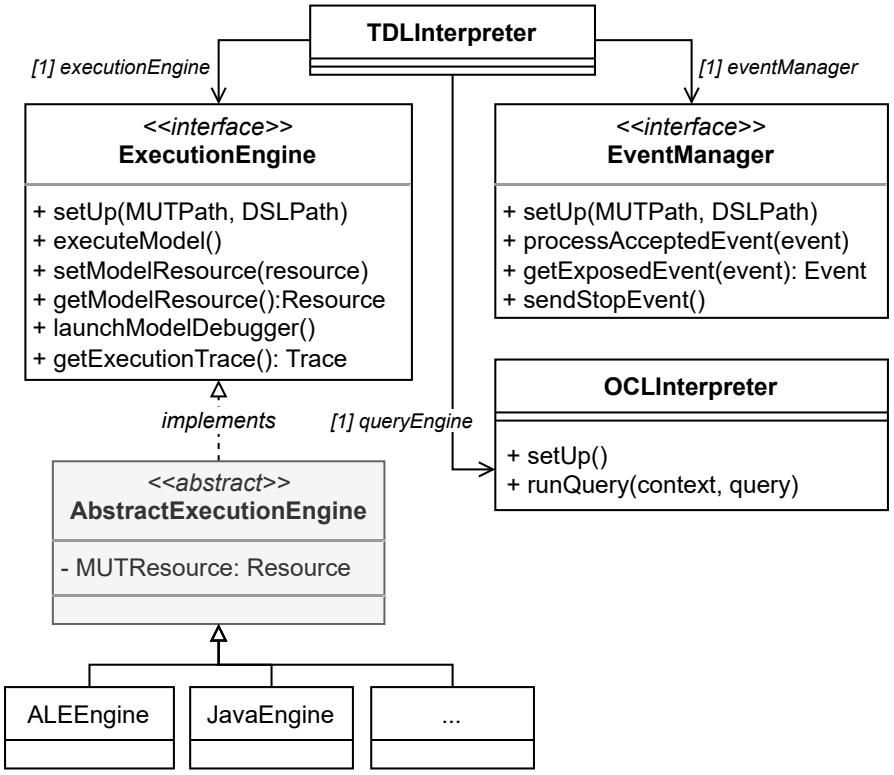


Figure 3.4: Class diagram showing the associations of the TDL Interpreter

The TDL Interpreter is connected to the execution engine to interpret the *model execution commands* used in a TDL test case and is connected to the query evaluator to interpret the *OCL queries* written in a TDL test case and to use the query evaluation result when required by a test oracle. These two connections enable our approach to run TDL test cases on 'non-reactive' models. The third connection is necessary for executing event-driven TDL test cases on 'reactive' models. The Event Manager must be configurable for a given reactive xDSL and allow external tools (e. g., testing tools) to interact with the xDSL's conforming models based on the xDSL's behavioral interface using two services: sending accepted event occurrences to a running model, and receiving its observable reactions as occurrences of the exposed events.

3.5.2.1 Overall architecture

The UML class diagram presented in Figure 3.4 shows the overall architecture of the TDL interpreter. As already illustrated in Figure 3.1, an execution engine uses the operational semantics of an xDSL to execute its conforming models and an event manager uses the behavioral interface of an xDSL and is connected to an execution

engine. To implement the operational semantics and the behavioral interface, different *metaprogramming approaches*—i. e., one or several metalanguages used in a particular fashion— can be used. Consequently, various execution engines and event managers may exist, each supporting a specific metaprogramming approach. To make the TDL Interpreter agnostic to this heterogeneity, we defined its required interfaces.

The ExecutionEngine interface can be used for setting up the execution engine based on the model under test (i. e., the model to be executed) and its conforming xDSL, for executing the model, for setting the model in a specific runtime state, and for getting its current state. Please note that the other two methods of the interface are required for other contributions of this thesis that will be explained in the next chapters. Following the terminology of popular frameworks such as EMF, we call *resource* the artifact that contains the model to load and execute. A resource may be a file or a URL to access the model remotely or a database connection. The ExecutionEngine interface is partially implemented by the AbstractExecutionEngine class, and then further specialized for each metaprogramming approach (in Section 3.7 examples of metaprogramming approaches are given). Therefore, our proposed approach is not restricted to a specific execution engine but can support all the existing ones. The EventManager interface comprises methods for setting up for a specific model and its conforming reactive xDSL, accepting an event to process on the model, retrieving an expected exposed event from the events exposed by the model, and stopping the communication with the model and releasing the resources. Finally, we also rely on a specific interface for the OCL Query Evaluator, here with the class named OCLInterpreter.

3.5.3 Test Execution Algorithm of the TDL Interpreter

In this section, we provide the details of the TDL Interpreter definition, mainly its test execution algorithm. Algorithm 1 shows the main loop, which requires as input a TDL package containing the set of TDL test cases to execute. For each test case, its test configuration must be activated first (line 3) using Algorithm 2. As can be seen, the path to the model under test and the name of the DSL are first retrieved from the annotations of the SUT component. Then, based on the connections between the gates, the required external components are instantiated and configured, including the Execution Engine, the OCL Interpreter, and the Event Manager.

Continuing with the main loop in Algorithm 1, after activating the test configuration, the test case behavior should be executed (line 4). The execution semantics of a behavior depends on its type. For instance, to execute a Message behavior (line 5), according to its source gate, the `argument` is treated differently. When the source gate belongs to a Tester Component, the `argument` is a request for the model

Algorithm 1: The TDL Interpreter main loop

```
   Input:
   package: the TDL package containing the TDL test cases to be executed
1  begin
2      foreach testcase ∈ package.testCases do
3          testcase.configuration.activate()
4          foreach behavior ∈ testcase.behaviors do
5              if behavior is Message then
6                  sourceGate ← behavior.source
7                  targetGate ← behavior.target
8                  if sourceGate.component.role is Tester then
9                      request ← behavior.argument
10                     targetGate.sendRequestToSUT(request)
11                 else if sourceGate.component.role is SUT then
12                     testOracle ← behavior.argument
13                     targetGate.assert(testOracle)
14             else if behavior is <other behavior types> then
15                 ...
```

under test (line 10), and when it belongs to a SUT Component, the `argument` is the expected result to be asserted (line 13).

Sending Requests to the SUT (shown in Algorithm 3): Depending on which gate of the SUT component is used for sending a request, the TDL Interpreter selects which external component (configured in Algorithm 2) should be used. Then, it checks whether the request can be accepted by the gate. Three cases are possible:

1. if the gate is a generic gate and the request is a model execution command (line 2), the configured engine should be used to run the command (line 3).

2. if the gate is an OCL gate and the request is an OCL query (line 4), the configured OCLInterpreter should be used to evaluate the query on the model (line 7). It should be noted that the query is evaluated on the model in its latest runtime state (line 6).

3. if the gate is a reactive gate and the request is an accepted event, the configured eventManager should be used to process the event.

Asserting the Expected Output (shown in Algorithm 4): The TDL Interpreter asserts whether an expected output data is equal to the real output data (i. e., the data received from the model under test). Depending on which gate of the SUT component is used for the assertion, the output data has different semantics:

Algorithm 2: Activating test case configuration

```
   Input:
   configuration: TDL test configuration to be activated
1  begin
2      MUTPath ← configuration.sutComponent.MUTPath
       DSLName ← configuration.sutComponent.DSLName foreach
       connection ∈ configuration.connections do
3          if connection between generic gates then
4              engine ←new ExecutionEngine()
5              engine.setUp(MUTPath, DSLName)
6          if connection between OCL gates then
7              OCLInterpreter ← new OCLInterpreter()
8              OCLInterpreter.setUp()
9          if connection between reactive gates then
10             eventManager ← new EventManager()
11             eventManager.setUp(MUTPath, DSLName)
```

Algorithm 3: Sending a request to the SUT

```
   Input:
   gate: the gate for sending requests to SUT,
   request: the request to be sent
1  begin
2      if gate is generic gate & request is modelExecutionCommand then
3          engine.runCommand(request)
4      if gate is OCL gate & request is OCL query then
5          query ← createQuery(request)
6          MUTResource ← getMUTResource()
7          OCLInterpreter.runQuery(MUTResource, query)
8      if gate is reactive gate & request is accepted event then
9          event ← createEvent(request)
10         eventManager.processAcceptedEvent(event)
```

- *generic gate*: the expected output is indeed a specific runtime state of the model under test. So the TDL Interpreter retrieves the current state of the model from the context of the engine (line 3), and then checks whether the model state is as expected (lines 4-7).
- *OCL gate*: the expected output is the expected query evaluation result, so it should be checked against the result generated by the OCLInterpreter (lines 9-12).
- *reactive gate*: the expected output is an exposed event expected to be received

Algorithm 4: Asserting expected output

```
   Input:
   gate: the gate for receiving data from SUT,
   expectedOutput: the expected output data to be asserted
   Output :
   verdict: the assertion result
 1 begin
 2     if gate is generic gate then
 3         currentState ← engine.context.resource
 4         if currentState == expectedOutput then
 5             verdict ← PASS
 6         else
 7             verdict ← FAIL
 8     if gate is OCL gate then
 9         queryResult ← OCLInterpreter.result if
            queryResult.equals(expectedOutput) then
10             verdict ← PASS
11         else
12             verdict ← FAIL
13     if gate is reactive gate & expectedOutput is exposed event then
14         expectedEvent ← createEvent(expectedOutput)
15         exposedEvent ← eventManager.getExposedEvent(expectedEvent)
16         if exposedEvent != NULL then
17             verdict ← PASS
18         else
19             verdict ← FAIL
```

from the model under test. Accordingly, the EventManager is requested to retrieve that event from the events exposed by the model (lines 14-15). If it retrieves nothing, the assertion fails (line 19).

In addition to the above-mentioned conditions, there are some specific cases that may lead to the interruption of the test case execution. This happens for instance when the test system sends a syntactically wrong OCL query to the SUT, or the exchanged event does not conform to the behavioral interface of the xDSL specified by the test configuration, or when the running external component throws some exception. In these cases, the TDL Interpreter interrupts the test case execution and sets the verdict to INCONCLUSIVE.

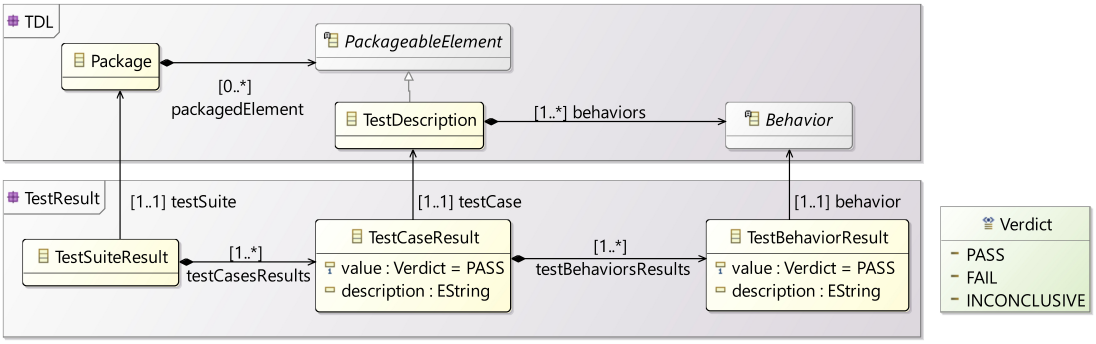


Figure 3.5: Test result metamodel

3.6 Test Result Reporter

At the end of the test execution, the *Test Result Reporter* component produces a report in a specific format presented in Figure 3.5. A test execution result is indeed captured as a TestSuiteResult for each test suite (i. e., a TDL Package element), comprising a set of TestCaseResult for each test case (i. e., a TDL TestDescription element). Each TestCaseResult has a value as PASS, FAIL, or INCONCLUSIVE, and a description about the test case execution result. It also comprises a set of TestBehaviorResult for each of the test case's containing Behavior elements, with a value and a description.

3.7 Tool Support

The proposed components of this chapter (shown in Figure 3.1) are implemented as part of the GEMOC Studio [29], a language and modeling workbench defined on top of the Eclipse Modeling Framework (EMF). TDL is also implemented using EMF technologies, making it easier to make both work together[3].

To transform Ecore to TDL, we used the ATL transformation language [82] and to transform behavioral interface to TDL, we used the implementation of Leroy et al. [93] for the behavioral interface definition (i. e., also a part of the GEMOC Studio) and we implemented the transformation in Java. For the TDL interpreter, we used Xtend [49] to implement the execution rules of the TDL operational semantics. To evaluate the OCL queries, we used the Eclipse OCL API [41] and for the Event Manager, we used an existing tool of the GEMOC Studio [93].

The GEMOC Studio supports several metaprogramming approaches for implementing the operational semantics of xDSLs. This includes Java-based languages (Kermeta [78], Xtend [49], and pure Java), the Action Language for EMF (ALE) [92],

[3] https://labs.etsi.org/rep/top/ide

xMOF [102], and a combination of a Java-based language with the MoCCML language [45]. Each of these approaches is supported by a dedicated execution engine. Hence, a significant part of the implementation of the TDL Interpreter is dedicated to managing different execution engines properly. For instance, the TDL Interpreter must read the xDSL definition in a GEMOC-specific `.dsl` file to discover the used metaprogramming approach, in order to start the correct execution engine accordingly. This should be noted that our current implementation supports Java-based languages and ALE among others.

After running a test suite, the execution result will be saved as an XMI file in the format given in Figure 3.5. In addition, we provide a user interface for the domain expert which visualizes the result. For example, Figure 3.6 shows a screenshot of the resulting tool. The source code is available on a public GitLab instance[4]. In the project explorer on the left, there are two projects, one containing the xArduino model (shown in Figure 2.2 on page 12) and another containing a TDL test suite written for it using the facilities presented in this chapter.

There are GUI icons in the toolbar and the menu bar allowing the user to choose an xDSL and run the TDL Library Generator for it. To use the tool for testing the xArduino model of Figure 2.2 (shown on page 12), we initially run the generator for the xArduino DSL and the generated TDL packages can be seen in Project Explorer (label 1). Here we show the execution of a test suite containing five sample TDL test cases (label 2), one of which is already described in Listing 3.9. We provided a view to report the test execution result along with some useful information for the user (label 3). As can be seen, the first test case failed because due to the defect of the model, the buzzer does not turn on and off for the second time (label 4).

3.8 Evaluation

We designed and performed an empirical evaluation of our approach to consider its *genericity* by answering the following research questions:

RQ#1 Does the approach provide testing facilities for xDSLs in which their abstract syntax is designed for *different domains*?

RQ#2 Does the approach provide testing facilities for both *non-reactive* and *reactive* xDSLs?

RQ#3 Does the approach provide testing facilities for xDSLs in which their operational semantics is implemented using *different metaprogramming approaches*?

[4] https://gitlab.univ-nantes.fr/naomod/faezeh-public/xtdl/-/tree/master/testing_framework

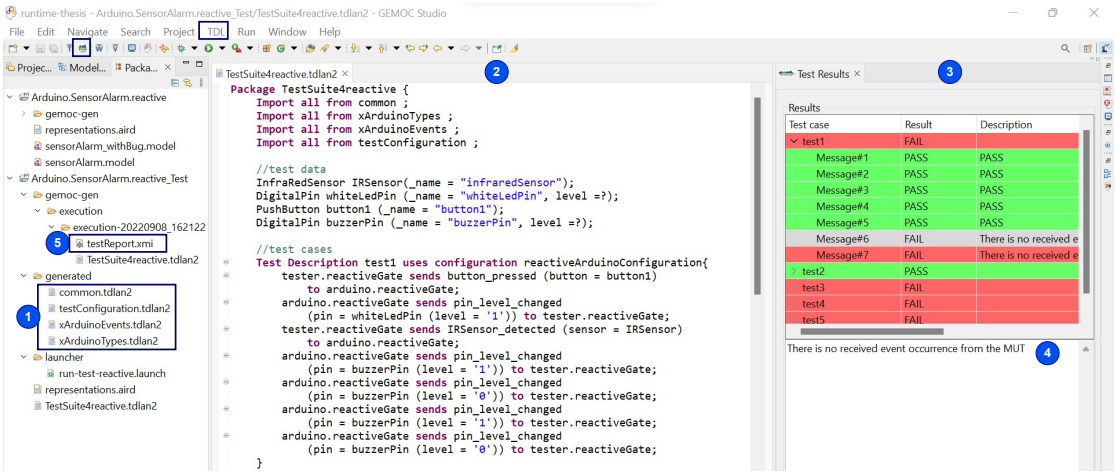


Figure 3.6: A screenshot of the provided testing tool running on the GEMOC Studio modeling workbench for the running example

3.8.1 Experiment Setup

Setup for RQ1. In the first research question, we aim to investigate whether the provided facilities can be used for xDSLs from different domains. Accordingly, we chose five xDSLs each covering a different domain:

- **xFSM**: A small language for designing Finite State Machines (FSM) for processing strings (taken from GEMOC official samples).
- **xArduino**: A language for simulating Arduino boards and their execution logic (described in Section 2.2).
- **xBPMN**: a representative of the *Microflow DSL*[5], a real-world xDSL introduced by the Mendix LCDP for modeling the application logic by data-flow modeling. It contains elements to perform CRUD operations on data objects, to show UI pages, to make choices, etc., and its graphical syntax is based on the Business Process Model and Notation (BPMN) standard.
- **xMiniJava**: A minimal implementation of Java based on the MiniJava project[6], allowing the definition of simple Java programs that can be executed directly by an execution engine rather than JVM. Note that it is not a typical xDSL and is defined just for experimental purposes as we will see in the following.

[5] https://docs.mendix.com/refguide/microflows
[6] https://www.cambridge.org/resources/052182060X

Table 3.3: Evaluation data for testing facilities

		Non-Reactive xDSLs				Reactive xDSLs	
		xFSM	xBPMN	xArduino	xMiniJava	xArduino	xPSSM
xDSL Size	Abstract syntax size (n. of EClasses)	3	39	59	76	59	39
	Operational Semantics size (LoC)	K3: 110 ALE: 90	ALE: 318	K3: 667 ALE:421	K3: 1042	K3:768	K3: 975
	Behavioral Interface size (n. of Events)	-	-	-	-	7	4
Tested Models	Number of tested Models	5	2	4	6	6	65
	Size range of tested models (n. of EObjects)	7-133	26-46	15-59	31-571	18-59	13-154
Test Artifacts	TDL Library size (LoC generated)	76	170	210	189	251	203
	Total n. of test cases	45	6	14	77	22	216
	Test case numbers range of test suites	7-16	2-4	2-4	4-25	3-4	1-81
	Size range of test suites(LoC)	50-157	33-50	25-64	33-188	30-132	25-1311

- **xPSSM**: A partial implementation of the Precise Semantics of UML State Machines (PSSM) [114] which supports modeling of discrete event-driven behavior.

Setup for RQ2. The objective of the second research question is to validate the practicality of the proposed testing facilities for both non-reactive and reactive xDSLs. Among the xDSLs introduced above, the operational semantics of xFSM, xBPMN, and xMiniJava are non-reactive, of xPSSM is reactive, and we used two implementations of xArduino semantics, one non-reactive and one reactive. Altogether, we used four non-reactive xDSLs and two reactive xDSLs for RQ2 as shown in Table 3.3.

Setup for RQ3. The third research question is about the support of the provided test execution facility from different metaprogramming approaches used for the implementation of xDSLs' execution rules. We considered that the execution rules can be implemented by Kermeta (K3) or ALE metaprogramming approaches.

As shown in Table 3.3, we used two different implementations of the xFSM and non-reactive xArduino semantics (K3 and ALE), and one implementation for the rest of the xDSLs (K3 or ALE).

We used the xFSM with Kermeta3 semantics [134] and with ALE semantics [133] from the GEMOC official samples. For the non-reactive xArduino, we used the open source project of the xArduino with Kermeta semantics [21], and then we implemented its semantics using the ALE language.

We designed the xBPMN language based on the definitions given in Section 2.2 and implemented its semantics using the ALE language. The xMiniJava implementation is taken from the GEMOC samples [137] and for the xPSSM, we used the implementation provided by the behavioral interface project [136], both with Kermeta semantics. We made minor modifications to these existing xDSLs to match the assumptions described in Section 2.2. The definition of all xDSLs is available in a public GitLab instance[7].

3.8.2 Evaluation Data

As presented in Table 3.3, the considered xDSLs have different sizes as the number of classes specified by their abstract syntax and the number of Lines of Code (LoC) of their operational semantics. For the two reactive xDSLs, their behavioral interfaces have different sizes as the number of events.

For each xDSL, we need a set of conforming models to be tested. Using each xDSL, we defined a couple of models in different sizes as their number of EObjects, including 5 xFSM models with 7 to 133 objects, 2 xBPMN models with 26 to 46 objects, 4 non-reactive xArduino models with 15 to 59 objects, and 6 xMiniJava models with 31 to 571 objects (taken from sample MiniJava programs[8]), 6 reactive xArduino models with 18 to 59 objects. For the xPSSM DSL, the PSSM standard provides a set of UML state machines, each with a small test suite for asserting that a PSSM implementation executes the models as expected, indeed in compliance with the standard [114]. We used a subset of them (60 models) which represent an event-driven behavior using solely state machines. In addition, we manually defined four larger state machines[9] for a total of 65 xPSSM models (60+5) with 13 to 154 number of EObjects.

[7] https://gitlab.univ-nantes.fr/naomod/faezeh-public/xtdl/-/tree/master/Language_Workbench

[8] https://www.cambridge.org/resources/052182060X/

[9] We used samples from: https://www.uml-diagrams.org/state-machine-diagrams.html

3.8.3 Evaluation Result

All the research questions are targeting the genericity of the proposed testing facilities, but from different perspectives: (RQ1) considering different domains; (RQ2) considering different model execution approaches (i. e., non-reactive and reactive); and (RQ3) considering different metaprogramming approaches. To answer them, we applied our proposed testing facilities to all considered xDSLs following the same process. First, we executed the *TDL Library Generator* component for each xDSL and it successfully generated a domain-specific TDL library for each of them. The number of LoC of each generated library is presented in Table 3.3. Most noticeably of all, it can be seen that, unsurprisingly, the size of the generated TDL Library increases with the size of the xDSL. For example, the TDL library of the reactive xArduino is the largest one with 251 LoC. This highlights one benefit of using our approach since the generated library provides all the TDL boilerplate code that the domain expert would otherwise write by hand. Therefore, the proposed approach reduces the cost of providing testing support for a given xDSL.

Second, using each generated TDL library, we wrote a set of TDL test cases for each considered model. In total, we wrote 45 test cases for 5 xFSM models, 6 test cases for 2 xBPMN models, 14 test cases for 4 non-reactive xArduino models, 77 test cases for 6 xMiniJava models, 22 test cases for 6 reactive xArduino models, and 216 test cases for 65 xPSSM models (60 of them are transformed from the standard PSSM test suites [114] to TDL and the rest are defined manually). As can be seen in Table 3.3, the size of the written test suites ranges from 25 to 1311 LoC.

Lastly, we executed the TDL test cases on the models using the *TDL Interpreter* component. For all the test cases, the test verdicts were set and the test results were reported using the graphical view provided by our tool. We also manually verified that we obtain the expected verdict for each test case. All the tested models, their test cases, and their test execution results are publicly accessible on a public GitLab instance[10].

In conclusion, we successfully used the proposed approach for all xDSLs whose abstract syntax represents *different domains* and whose execution semantics was implemented in different styles (non-reactive and reactive) and also in different metaprogramming approaches (K3 and ALE). Therefore, we can conclude that our approach is not tied to only one specific xDSL, and thus satisfies the *genericity* aspect.

[10] https://gitlab.univ-nantes.fr/naomod/faezeh-public/xtdl/-/tree/master/Modeling_Workbench

3.8.4 Threats to Validity

In evaluating the genericity feature, we only considered six languages, so there is an external threat that the testing facilities might not work as expected for other modeling languages. Additionally, we defined our testing facilities considering the GEMOC Studio as a reference for the xDSL implementation. As there are also other language workbenches [51], additional studies are required to validate the portability of the proposed facilities.

Our proposed approach aims to support domain experts in writing and executing test cases for their executable models. To validate its usability for the domain expert, a user study should be performed. Accordingly, a threat exists regarding the approach usability and we consider it for our future work. However, as our approach uses TDL which is a standard testing language particularly defined for non-technical testers, and as we support using the domain concepts in writing TDL test cases, we tried to take the usability feature into account.

3.9 Conclusion

Providing testing facilities for any given xDSL is a challenging task concerning the diversity of xDSLs. This diversity originates from both, the domain described by the xDSL abstract syntax and the approach used for the implementation of its semantics. This chapter introduced our contributions for the definition and execution of test cases for executable models defined by metamodel-based xDSLs. We used the TDL standard testing language for describing test cases and provided solutions to specialize TDL for testing executable models conforming to xDSLs. Indeed, we proposed a TDL library generator that generates a domain-specific TDL library for a given xDSL, allowing the domain expert to write test cases for testing the executable models conforming to it. We also provided an interpreter for TDL to execute TDL test cases on executable models. Our evaluation on several various xDSLs demonstrated that the provided facilities realize the genericity aspect. In conclusion, we observed that our generic test definition and execution facilities for xDSLs advance the testing tool support for existing as well as emerging xDSLs.

Discussions and Improvements. In this chapter, we explored the proposed test case definition and execution facilities on a certain set of xDSLs, and we tested their conforming models in some specific ways; based on their runtime state, their exchanging events, and using OCL queries. In chapter 6 on page 129, we will explain the limitations of the facilities for other situations and introduce possible improvements for them. We also noted in Section 3.5.1 on page 63 that the TDL interpreter does not currently support all the elements of the TDL standard abstract

syntax. By completing the definition of the TDL interpreter, we can write and execute more complex TDL test cases to control different features of the model under test.

Moreover, being able to write and execute test cases is the first step in testing models and it is not adequate. The written test cases must be evaluated to make sure they are good enough at exercising the majority of the model elements and at detecting their potential faults. These concerns are discussed in the next chapter.

Chapter 4

Test Quality Measurement

4.1 Introduction

When test cases do not find any bug, while it may verify the correctness of the tested model (in the best case), it may also highlight weaknesses of the test suite (in the worst case). Therefore, there is a need for evaluating the quality of the test suites to efficiently test a system. In the realm of programming languages, coverage computation and mutation analysis are two popular means of test quality measurement [12]. The former measures how much of the system under test is exercised by a given test suite [12], while the latter analyzes the ability of the test suite in finding potential faults [79].

Although test quality measurement techniques have existed for a long time for GPLs, to our knowledge, they are still understudied when it comes to xDSLs. One possible solution is to develop them "from scratch" for each and every xDSL. For instance, one can design a coverage metric tailored only for a State Machines xDSL—where a *state* is considered covered when it is reached during the execution of a state machine [147]—and can create associated tools. But undertaking this effort for each new xDSL is a costly and error-prone task, which adds up to the cost of making the xDSL itself. Regarding mutation analysis, while a recent approach aims to provide mutation analysis for xDSLs [62], this approach is incomplete as it is not yet able to actually run test cases on the generated mutants. Therefore, more generic solutions, as available for other concerns for xDSLs, are considered beneficial.

To properly measure the quality of the TDL test suites written for models, we offer both *Coverage Computation* and *Mutation Analysis* techniques for xDSLs as part of our proposed testing framework. For coverage computation, we propose a generic approach in which the coverage of a model's test suite is calculated by analyzing the model's execution traces. It also allows the language engineers to

customize the generic coverage measurements for their xDSLs. In Section 4.2 on page 80, more details are provided.

For mutation analysis, we provide an integration of our testing facilities with a generic model mutation framework named WODEL proposed by Gómez-Abajo et al. [62]. WODEL allows the definition of mutation operators for a given metamodel-based xDSL which are then used by WODEL mutant generator to produce mutants out of any conforming model. Our integration handles the execution of a model's test suites (i. e., written by our testing facilities) on the mutants generated by WODEL for the model, and produces the result such as the mutation score, and list of the mutants killed by each test case, among others. More details are presented in Section 4.3 on page 89.

4.2 Coverage Computation

Regardless of the xDSL used for the definition of an executable model, every model can be formally defined as a specific kind of graph, the so-called *type graph*, in which model elements are defined as nodes, and different types of edges exist to specify containment, inheritance, and cross-references [25]. When executing a model, the result can systematically be captured in an execution trace, using a fixed and generic format, which keeps track of the model's exercised elements—i. e., nodes of the corresponding type graph. Based on this perspective, and through an analysis of the xDSL definition itself, it is apparent we can adapt the *node coverage* metric—from structural graph coverage criteria [12]—for the context of xDSLs, hence reasoning about the model coverage in a generic way. Indeed, we can define a new coverage metric that generally considers models' elements as software components to be covered.

In this section, we introduce a new coverage computation approach for the context of xDSLs. It can be used for computing the coverage measures for any model, regardless of the xDSL used for its definition. Figure 4.1 displays an overview of the proposed approach which has the same roles as the previous chapter (shown in Figure 3.1 on page 49): a language engineer (at the top) who defines an xDSL, and a domain expert (at the bottom) who defines models (using the xDSL) and test cases for them, and wishes to evaluate the quality of his/her written test cases in terms of coverage.

Using the *Coverage UI* component (label 1), the domain expert can request for computing the coverage of his/her written TDL test cases. This component asks the *TDL Interpreter* (label 2, i. e., already proposed in Section 3.5 on page 62) to run test cases and to generate the execution trace of the model under test. The *xModel Coverage Computation* component (label 3) then generates the coverage matrix of each model's test case using two sources of information. First, from the

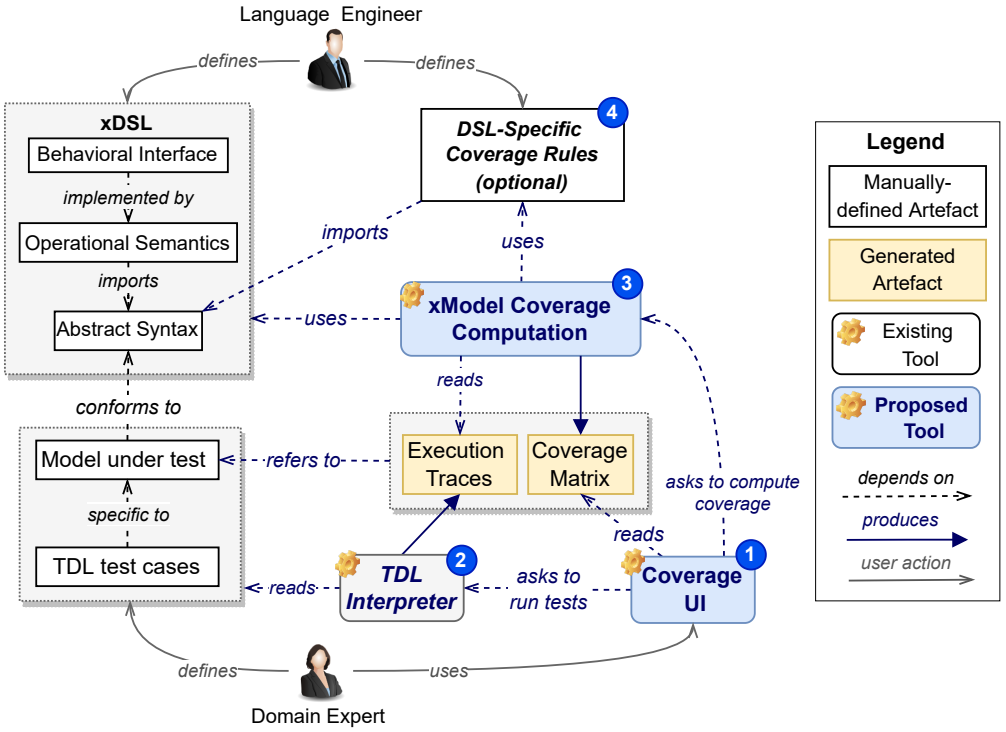


Figure 4.1: An overview of coverage computation approach

definition of the given xDSL, it recognizes which classes of the abstract syntax are used by each execution rule of the operational semantics. This is required to recognize what are the "traceable" elements of the model, i. e., elements whose execution may be captured in the trace. Second, it analyzes the execution trace of the tested model to extract the model's elements that are captured in the trace, meaning that they are *covered* by the test case and the "traceable" elements that are not captured in the trace are *not covered*.

In addition to what can be considered in a generic coverage metric, specific coverage aspects may be required for particular xDSLs. To this end, the approach allows the language engineers to optionally define specific coverage rules for their xDSL (label 4). Accordingly, the *xModel Coverage Computation* component uses such rules, if they are available, to update the generated coverage matrix according to the specific needs of the xDSL. At the end, the *Coverage UI* component visualizes the computed coverage in a user interface for the domain expert and can also save the coverage matrix in a persistent file.

4.2.1 Constructing the Coverage Matrix

Given a TDL test case executed on a model conforming to an xDSL, we need information about the model execution to construct its coverage matrix which can be accessed from two main sources: the definition of the xDSL and the model execution trace.

4.2.1.1 Analyzing the xDSL definition

As explained in Section 2.2.3 on page 13, given an xDSL, the execution of a conforming model is performed by calls to the execution rules of the xDSL operational semantics. Each execution rule uses specific classes of the xDSL's abstract syntax. This means that, when running a model, the execution of its individual elements will be captured in a trace only if there is at least one execution rule defined for either a direct or inherited type of the element. Therefore, by analyzing the definition of an xDSL, we can identify the classes of its abstract syntax for which instances can be considered traceable, and thus whose coverage by a test case can be detected using an execution trace. Algorithm 5 shows this analysis with an xDSL as input and a list of classes namely traceableTypes as output. Its output will be used for the coverage computation of the models which are defined by its input xDSL.

Algorithm 5: Finding the traceable types of an xDSL

Input:
xDSL.syntax: the abstract syntax of the xDSL,
xDSL.semantics: the operational semantics of the xDSL
Output :
traceableTypes: classes of the xDSL's abstract syntax for which the execution of their objects can be traced

```
1 begin
2     foreach rule ∈ xDSL.semantics do
3         traceableTypes.add (rule.class)
      // Checking inheritance relationships
4     foreach class ∈ xDSL.syntax do
5         if class ∉ traceableTypes
6         ∧ class.allSuperClasses → exists (c|c ∈ traceableTypes) then
7             traceableTypes.add (class)
```

4.2.1.2 Initializing the coverage matrix for the models' tests.

After running a test case on a model, we compute its initial coverage using the model's execution trace. Please note that coverage can be only computed for passed or failed test cases (not the ones with an inconclusive result). We described in Section 2.2.4 on page 21 that such a trace is a sequence of called execution rules on the elements of the model [31, 32] (also shown in Figure 2.7 on page 23). Therefore, by analyzing the trace, we can extract the model's elements covered by the test case.

For example, if we run the TDL test case of listing 3.9 on page 64, on the xArduino model of Figure 2.2 (shown on page 12), it results in running the xArduino model itself by calling rules of the reactive xArduino semantics (part (**b**) of Figure 2.5 on page 19) as follows. When the test case sends a request for pressing `button1` (sending button_pressed(button1) event to the xArduino model), first the press(button1) rule is called, which results in a set of consecutive calls: execute(sketch), execute(if), and execute(White LED=1) that turns on the `White LED` by calling changeLevel(whiteLEDPin (level=1)) because the `button1` is pressed (it also resulted in exposing an occurrence of the pin_level_changed event for the whiteLEDPin).

Next, the test case requests to put the `infrared sensor` in the state of detecting an obstacle (sending IRSensor_detected(infrared sensor) to the xArduino model). This results in a call of detect(infrared sensor) which triggers the sequence execute(if), execute(buzzer=1) that turns on the alarm (by calling the changeLevel(buzzerPin (level = 1)) which then exposes an occurrence of the pin_level_changed event for the buzzerPin), execute(buzzer=0) that turns off the alarm (by calling the changeLevel(buzzerPin (level = 0)) which then exposes an occurrence of the pin_-level_changed event for the buzzerPin), execute(buzzer=0), that must turn on the alarm for the second time but due to the defect it does not, and execute(buzzer=0).

This set of calls is captured in an execution trace of the xArduino model as shown on the top of Figure 4.2 on page 84. Using this trace, we can construct the initial coverage matrix of the test case of listing 3.9 on page 64. As displayed on the bottom of Figure 4.2, we consider the elements captured by the trace as "covered" (highlighted in green) and the rest (highlighted in yellow) will be examined in the next steps of coverage computation described in subsequent.

4.2.1.3 DSL-specific coverage rules

In addition to what can be considered in a generic coverage metric, specific coverage aspects may be required for particular xDSLs [108]. So far, we considered an element is covered solely based on what we were able to observe in the execution, i. e., if it was captured in a trace. However, this information may also allow deducing

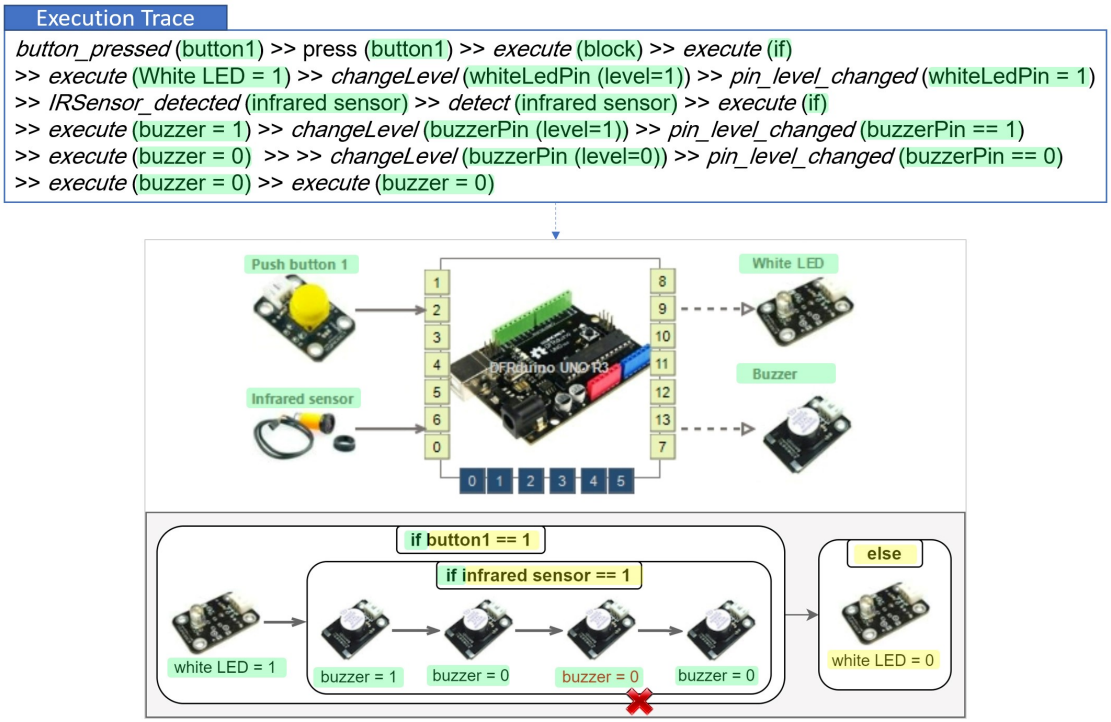


Figure 4.2: Coverage of the xArduino model of Figure 2.2 on page 12 by the TDL test case of listing 3.9 on page 64 based on its execution trace (covered elements are highlighted in green, and yellow-highlighted elements will be examined in the next steps of computation)

that other elements (e. g., referenced, contained by elements in the trace) can be considered as covered as well.

To provide this customizability, our approach optionally allows a language engineer to define a set of *DSL-specific coverage rules* for a given xDSL (shown at the top right corner of Figure 4.1 on page 81). More specifically, we propose a dedicated metalanguage for defining such rules whose concepts are presented in Figure 4.3 on page 85. Given the abstract syntax of an xDSL in the form of a metamodel, a DomainSpecificCoverage can be defined for different Contexts each pointing to a metaclass of the xDSL's abstract syntax. For each Context, several Rules can be defined and we are currently considering two families of rules:

- **Inclusion rules**: a covered object, may induce that other objects are covered as well (see CoverageOfReferenced and CoverageByContent rule types).
- **Exclusion rules**: an object is ignored from coverage computation under a certain condition (see Ignore and ConditionalIgnore rule types).

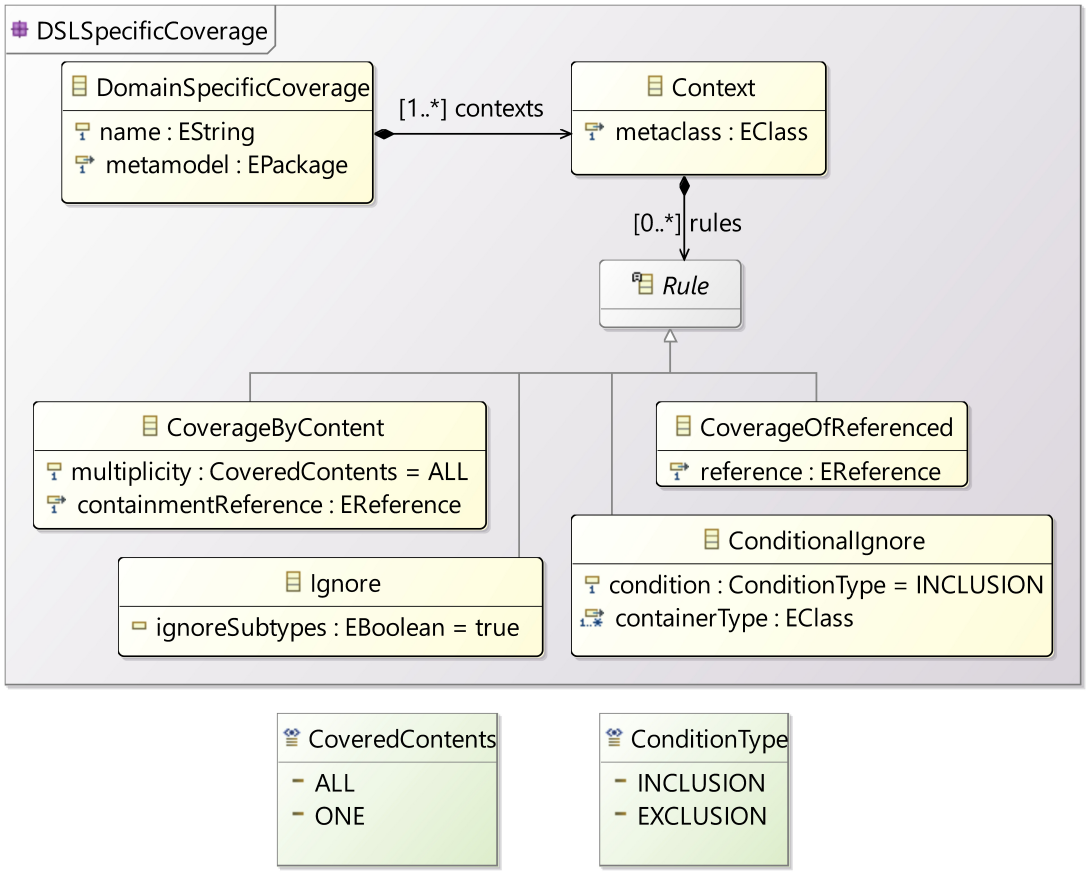


Figure 4.3: DSL-specific coverage metamodel

Given an object conforming to a Context (directly or by inheritance), each type of rule acts as follows:

CoverageOfReferenced. From the coverage of the given object, we infer the coverage of its referenced objects (i. e., the value of its reference feature). Accordingly, the type of the reference will be added to the list of `traceableTypes` (i. e., output of Algorithm 5 on page 82).

CoverageByContent. Inferring the coverage of the given object from the coverage of its contained objects (i. e., the value of its containmentReference feature). The object is covered if:

- `multiplicity = ALL`: all of its contained objects are covered.
- `multiplicity = ONE`: at least one of its contained objects is covered.

This rule also updates the list of `traceableTypes` by adding the metaclass of the Context to it.

Ignore. The object will be ignored from coverage computation, by considering it as "not-traced", except when the rule specifies not to ignore it if it conforms to the subclasses of the context (ignoreSubtypes= false).

ConditionalIgnore. The object will be ignored from coverage computation, by considering it as "not-traced", when it is contained by an object that:

- `condition = INCLUSION`: conforms to one of the containerType classes.
- `condition = EXCLUSION`: does not conform to any of the containerType classes.

These rules are applied in order repeatedly until a fixed point is reached i. e., until the coverage matrix becomes steady.

For example, Listing 4.1 shows some of the rules we have defined for the Arduino xDSL. The CoverageByContent rule specifies that a Block object is covered if at least ONE of its contained Instruction elements is covered. According to the definition of the xArduino event-driven semantics ((**c.1**) and (**c.2**) parts of Figure 2.6 on page 22), there is no execution rule for the Expression class. Indeed Expression objects are evaluated inside other rules such as the execute(If). According to this information, we defined a CoverageOfReferenced rule specifying that whenever an If object is covered, its condition that is an Expression is also covered. We also defined an Ignore rule to ignore instances of Module from coverage computation as they are just representatives of physical elements.

4.2.1.4 Finalizing the coverage matrix for the models' tests

At the last step of coverage computation, we identify "not-covered" objects as follows. Given an object with an unspecified coverage status, it is "not-covered" if its type is traceable—contained in the traceableTypes list— and "not-traced" otherwise. Please note that we computed the traceableTypes in the previous steps, by analyzing the xDSL operational semantics (algorithm 5 on page 82) and running the xDSL-specific coverage rules if we have any (Section 4.2.1.3 on page 83).

Finally, we generate a complete coverage matrix for the whole test suite of the model by merging the coverage matrices produced for each of its test cases.

```
DomainSpecificCoverage ArduinoCoverageRules{
    Import metamodel arduino
    Context Block{
        CoverageByContent{
            containmentReference instructions
            multiplicity ONE
        }
    },
    Context If{
        CoverageOfReferenced {reference condition}
    },
    Context Module{
        Ignore {ignoreSubtypes true}
    }
}
```

Listing 4.1: Examples of Arduino-specific coverage rules

4.2.1.5 Generating a coverage matrix for the running example

An excerpt of the result produced by each of the above-mentioned steps for some of the objects of the xArduino model of Figure 2.2 (shown on page 12) is provided in Table 4.1. As can be seen:

- the `button1` object is considered as covered after trace analysis (Step 1), but is then ignored after updating the coverage matrix by the Arduino-specific coverage rules (Step 2).
- the `if` object is covered based on the trace analysis (Step 1).
- the `button1 == 1` Expression does not have any status at first (Step 1) but it is then updated to covered after running the Arduino-specific coverage rules (Step 2) because when the `if` object is covered, its referenced expression element must be considered as covered.
- the `white LED = 0` ModuleAssignment is not covered by the test case (Step 3)

At the end, the final coverage matrix is equivalent to the content of Table 4.1 modulo columns 3 and 4.

4.2.2 Definition of Artefacts

To preserve the genericity of our proposed approach, this section introduces a generic definition for the coverage matrix of the models' tests, presented in Figure 4.4. Coverage can be computed after running a test case on a model and once a test case is executed, two artifacts will be generated first: the test execution result and

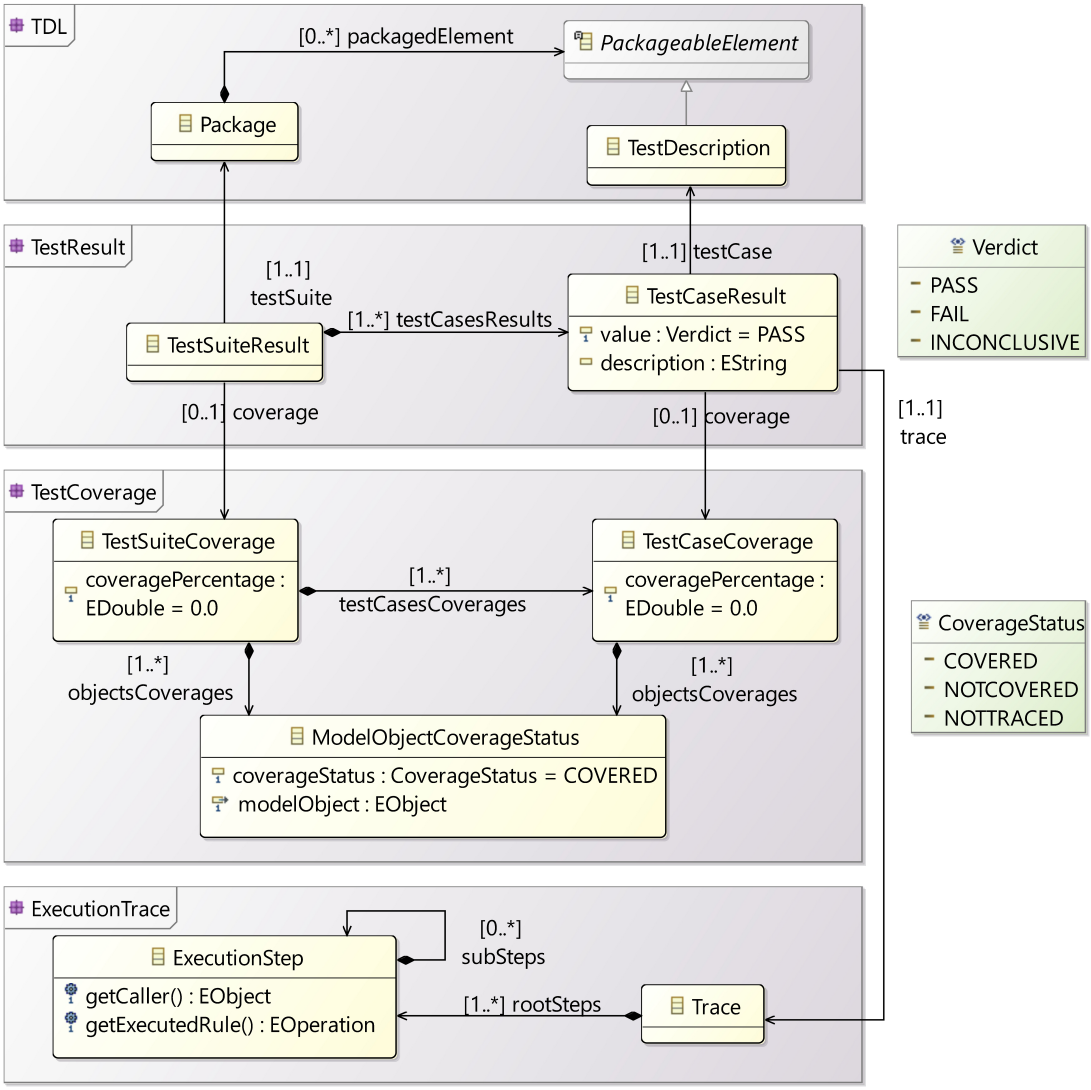


Figure 4.4: Definition of artifacts

Object	Type	Step1 Initial Coverage	Step2 Updated Coverage	Step3 Final Coverage
button1	PushButton	**Covered**	**NotTraced**	NotTraced
if	If	**Covered**	Covered	Covered
button1==1	Expression	-	**Covered**	Covered
white LED=0	Module-Assignment	-	-	**Not-Covered**

Table 4.1: An excerpt of the coverage computation for the running example (changes of the step in bold)

the model execution trace. We keep these two artifacts related to each other by connecting the test result metamodel of Figure 3.5 on page 71 to the execution trace metamodel of Figure 2.7 on page 23 through adding a reference from the TestCaseResult to the execution Trace of its tested model.

Once coverage is computed for an executed passed or failed test suite, a TestSuite-Coverage is generated which includes one TestCaseCoverage for each of its executed test cases. Both of them have a list of ModelObjectCoverageStatus instances, each specifies the coverage status of one object of the model for the test case/test suite. The coverage status for the elements is either COVERED, NOTCOVERED, or NOTTRACED.

4.3 Mutation Analysis

To provide mutation analysis in a generic model testing approach, four features are required: (1) definition of mutation operators for a given xDSL; (2) a process to generate mutants out of a given model under test (i. e., conforming to the considered xDSL) by applying the mutation operators; (3) a way to execute the model's test suite on each generated mutant; and (4) a way to calculate the mutation score for the test suite. Recently, a framework named WODEL-Test was proposed by Gómez-Abajo et al. which is able to support most of these features [62]. More specifically, WODEL-Test allows a language engineer to define mutation operators for her/his xDSL if the abstract syntax is provided as a metamodel (feature 1). Then, it automatically generates mutants for the models conforming to that xDSL by applying the defined mutation operators (feature 2).

However, WODEL-Test does not provide any testing facility and thus fails at providing feature (3). It indeed assumes there is an existing testing framework for the given xDSL which allows writing test suites for the conforming models and

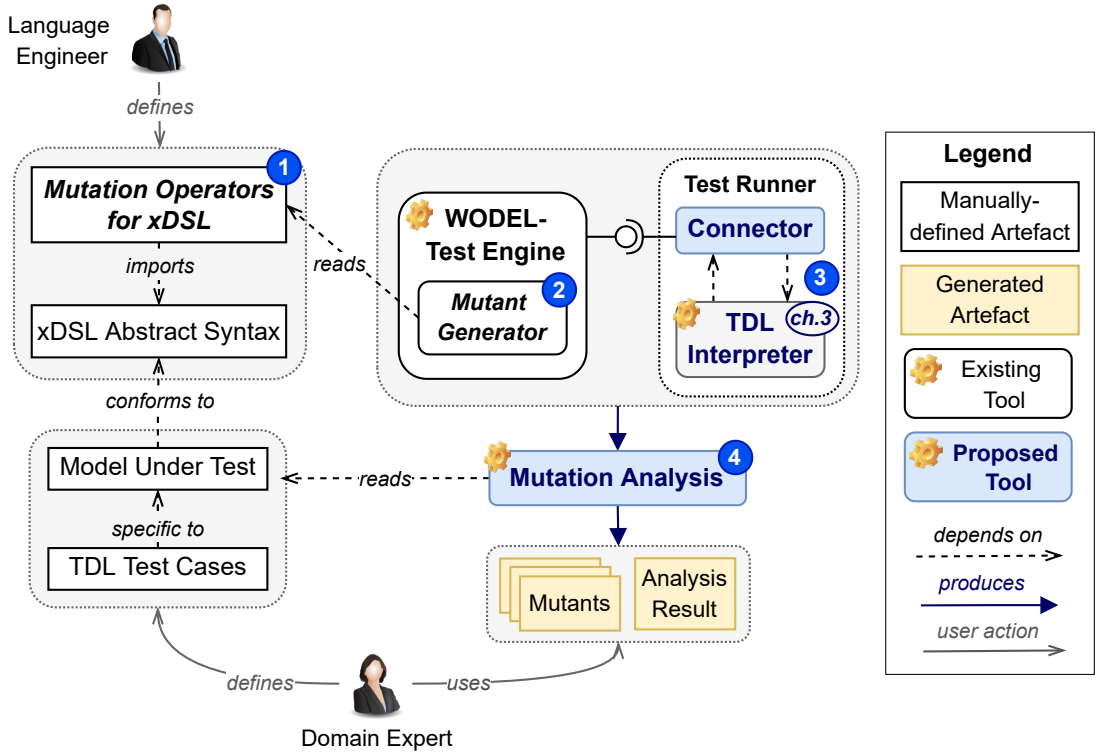


Figure 4.5: An overview of the integration of the TDL Interpreter with WODEL-Test [62], resulting in a mutation testing tool

provides an interface to run such test suites and get the result. Based on this assumption, WODEL-Test generates an environment for the domain experts to run their written test suites on the generated mutants and to get their mutation scores (feature 4). As our proposed testing facilities realizes this assumption, we can offer a complete mutation testing framework for any xDSL through integration with WODEL-Test.

Figure 4.5 shows how we achieved this integration, where the same roles and artifacts as our testing approach (Figure 3.1 on page 49) are involved. There is only one additional task for the language engineers to define mutation operators for their xDSLs (label 1). The WODEL *Mutant Generator* uses these operators to produce mutants from the conforming models (label 2).

As mentioned above and shown in Figure 4.5, WODEL-Test requires a test runner (label 3) implementing a specific interface that determines how to run a test suite on a mutant and how to decide whether the given mutant is killed by the test suite. Accordingly, we have connected our TDL Interpreter to the WODEL-Test engine by defining a connector that implements the interface and performs three

operations:

1. it receives a TDL test suite and a mutant from the WODEL-Test and runs the test suite on the mutant using the TDL Interpreter.

2. it receives the test execution result from the TDL Interpreter and sets the mutant as 'killed' if there is at least one test case in the test suite that is failed on the given mutant.

3. it provides the final mutation testing results in conformance to the WODEL-Test result templates.

It is worth mentioning that for this integration, we added some extra features in our TDL Interpreter component. As described in Section 3.4 on page 53, the TDL Interpreter runs a TDL test case on the model that is persisted in the path specified in the test configuration of the test case (the value of the MUTPath in listings 3.6 and 3.7 on pages 60 and 61). Consequently, to execute the test case on another model, we need to modify the test configuration. However, for mutation testing, a TDL test case must be run on several models i. e., the original model and the mutants generated for it, without modifying the test case definition—including the test configuration. To this end, we provide an optional service in the TDL Interpreter to be able to run a test case on a specific model while ignoring the model path specified in the test configuration.

The result of our provided integration is realized by the *Mutation Analysis* component (label 4). Using this component, the domain experts can measure the ability of their TDL test suites in detecting potential faults in the models.

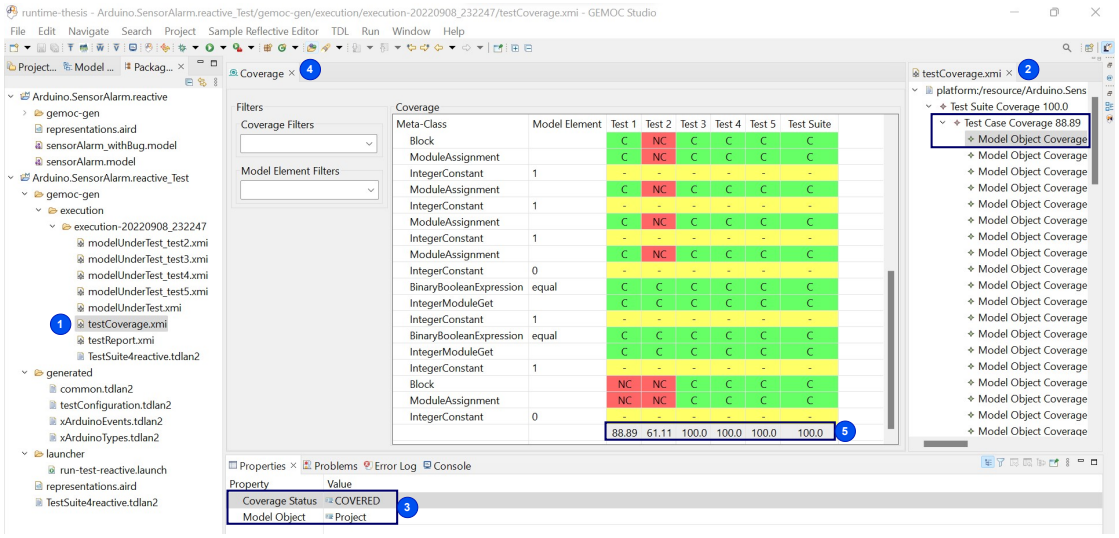
4.4 Tool Support

We implemented our proposed quality measurement features as part of the GEMOC Studio [29]. For generating the execution trace of an executed model, we used a generic execution trace management tool for xDSLs proposed by Bousse et al. [31, 32] which is also part of the GEMOC Studio. All the proposed components (the *xModel Coverage Computation*, the *Coverage UI*, and the *Mutation Testing* components), are implemented in Java and are connected using the Eclipse extension point mechanism. For the *DSL-Specific Coverage Rules* metalanguage, its metamodel is defined in Ecore [132], its concrete syntax is defined by Xtext[1], and its semantics is implemented in Java.

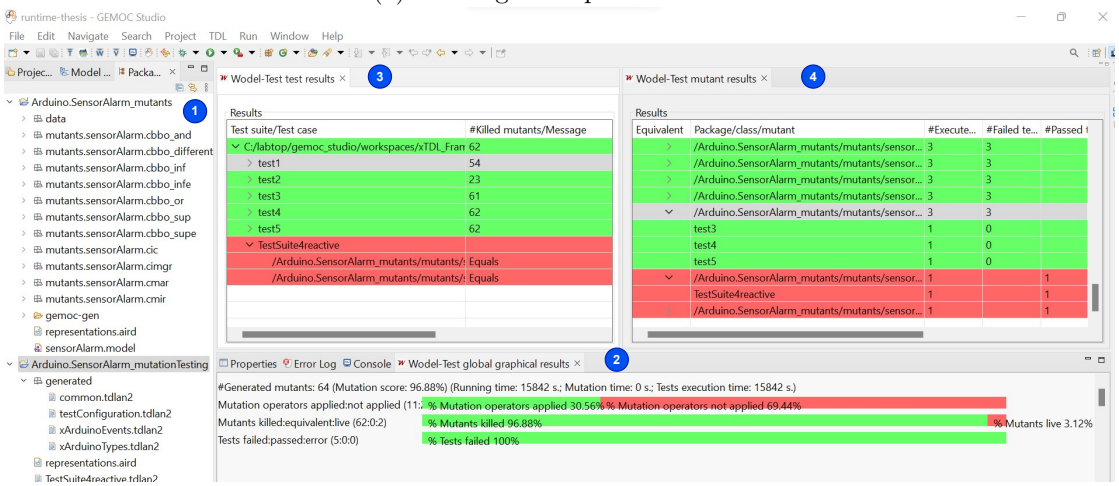
For the *Mutation Analysis*, as WODEL-Test framework is also implemented using EMF technologies [62], we easily integrated it into our testing framework.

[1] https://www.eclipse.org/Xtext/

More specifically, we implemented a connector in Java (shown in Figure 4.5 on page 90) which connects the WODEL-Test engine to our TDL Interpreter.



(a) Coverage Computation Tool



(b) Mutation Analysis Tool (using WODEL-Test facilities [62])

Figure 4.6: Screenshots of the provided quality measurement tools running in the GEMOC studio modeling workbench for the running example

Figure 4.6 displays two screenshots of the provided facilities running in the GEMOC studio modeling workbench for the running example. A screenshot of the coverage tool is shown in Figure 4.6(**a**) and its source code is available on a public GitLab instance[2]. In the project explorer on the left, there are two projects,

[2] https://gitlab.univ-nantes.fr/naomod/faezeh-public/xtdl/-/tree/master/coverage_computation

one containing the xArduino model (shown in Figure 2.2 on page 12) and another containing a TDL test suite written for it using our testing facilities presented in Chapter 3 on page 47. The coverage matrix can be persisted as an XMI file conforming to the format presented in Figure 4.4 (on page 88) upon the request of the user—the user can select the related option in the run configuration (label 1). For each executed test case, a copy of its model under test is also saved and its objects are referenced by the generated 'testCoverage.xmi' file (labels 2 and 3).

We provided a graphical view for displaying the coverage measures computed for the test cases as well as for their test suite (label 4). For each element of the model under test, it shows its coverage status for all the tests, green for COVERED, red for NOTCOVERED, and yellow for NOTTRACED elements. Moreover, the last row (label 5) provides the percentage of the traceable elements covered by each test case and also by the whole test suite (i. e., 100 %). The user can also use two filter options, one to find all the elements with a specific coverage status (Coverage Filters on the left), and another to find the coverage status for a specific type of the elements (Model Element Filters on the left).

Figure 4.6(**b**) shows how mutation analysis appears in the tool. Its source code is available on a public GitLab instance[3]. Here, we analyzed a TDL test suite (containing five test cases) for the correct version of the xArduino model of Figure 2.2 on page 12. Note that we first defined 36 mutation operators for the xArduino DSL which will be introduced in Section 4.5.1 on page 94. Under the xArduino project (label 1), one folder per applied mutation operator exists, each containing mutants generated by WODEL-Test through applying that operator. The results of running the TDL test suite on both the original model and the generated mutants are as follows. The global result (label 2) reports that 64 mutants are generated by applying 30.56 % of mutation operators (11 out of 36). Among them, 62 are killed by the TDL test suite, so its mutation score is 96.88 %. The tool also reports the mutants killed by each test case and the alive mutants (label 3) as well as the test suite execution result for each mutant (label 4).

4.5 Evaluation

In this section, we aim at evaluating both proposed test quality measurement facilities. The main objective of our proposed testing framework is *genericity* regarding its supported xDSLs. Accordingly, our first research question is:
RQ1: How much genericity is supported by the approach in providing test quality measurement features for xDSLs?
Moreover, for the proposed coverage computation approach, we aim to answer two

[3] https://gitlab.univ-nantes.fr/naomod/faezeh-public/xtdl/-/tree/master/mutation_analysis

		Non-Reactive xDSLs		Reactive xDSLs		Total
		xFSM	xMiniJava	xArduino	xPSSM	
Tested models	Number of tested models	5	6	6	65	82
	Size range of tested models (n. of EObjects)	7-133	31-571	18-59	13-154	7-571
Test Artifacts	Total number of test cases	45	77	22	216	360
	Test case numbers range of test suites	7-16	4-25	3-4	1-81	1-81
Coverage	Number of DSL-specific coverage rules	-	4	8	13	25
	DSL-specific coverage size (LoC)	-	26	50	75	151
	Number of test suites with computed coverage	45	77	22	216	360
	Range of computed test suites' coverage	100%	98.08-100%	100%	100%	-
Mutation Analysis	Number of mutation operators	5	113	36	30	184
	Number of generated mutants	289	181	394	12,087	12,951
	Number of killed mutants	194	120	375	9,989	10,678

Table 4.2: Evaluation data for coverage computation and mutation analysis

further questions:
RQ2: How much customization is needed in order to have the intended coverage computations for xDSLs?
RQ3: To what extent is the result of the coverage computation component valid?

We performed an empirical study of our proposed approach to answer the research questions which is presented in this section.

4.5.1 Experiment Setup

Setup for RQ1. For the first research question, we aim to investigate whether the quality measurement facilities can be used for different xDSLs. Accordingly, from the xDSLs used for the evaluation of our testing facilities in Section 3.8 on page 72, we chose xFSM, xMiniJava, reactive xArduino, and xPSSM because they are from different domains, their operational semantics is implemented in different ways (i. e., xFSM and xMiniJava are non-reactive and xArduino and xPSSM are reactive) and they have different sizes (as written in Table 3.3). We already introduced them in Section 3.8.1 on page 73.

Since our objective here is to evaluate the quality of the test suites using the proposed approaches, we used all the TDL test suites written in Section 3.8 on page 72 for the models conforming to each xDSL. Altogether, 360 test cases for 82 executable models (Table 4.2).

Moreover, for evaluating the mutation analysis facility, we need a set of mutation operators for each considered xDSL as well as mutants generated out of the models using the operators. As Table 4.2 presents, using WODEL language [61], we

defined 5 mutation operators for the xFSM, 113 for the xMiniJava, 36 for the xArduino (introducing faults only in the Sketch part of the models and ignoring the physical-related concepts), and 30 for the xPSSM (based on previous work on state machine mutation [55, 62, 95, 119, 128]) —cumulatively 184 mutation operators for our considered xDSLs. We then applied the operators on each considered model and WODEL generated a total of 12,951 mutants for our 82 models.

Setup for RQ2. The second research question is targeting the required customization for coverage computation when using the proposed approach for various xDSLs. Therefore, for RQ2, we also consider the four xDSLs chosen in the setup for RQ1, aiming at investigating which one of them needs customization and how the proposed approach allows it.

Setup for RQ3. One way to answer RQ3 is to compare our coverage computation component with an existing coverage tool. As xMiniJava is a Java-like xDSL, each xMiniJava model is indeed a Java program and test cases of the xMiniJava models can be defined as JUnit tests for the equivalent Java programs. So we can compare our coverage computation approach with an existing Java coverage tool. For this comparison, we have chosen CodeCover [117] as it is an open-source coverage tool supporting JUnit tests of Java programs [4]. Among different coverage metrics provided by CodeCover, we use *statement coverage* as it is the closest to our metric. CodeCover uses source code instrumentation approach to compute statement coverage [117].

We transformed test cases of xMiniJava models—according to Table 4.2, 77 tests for six xMiniJava models—to JUnit tests for equivalent Java programs. We reused the Java programs provided by the MiniJava project[5].

Evaluation data. The evaluation data is accessible from a GitLab repository[6].

4.5.2 Evaluation Result

Answering RQ1. In the first research question, we aim to evaluate whether the coverage computation and the mutation analysis facilities can be used for evaluating the quality of the test cases written for different models defined by various xDSLs. To answer RQ1, we used the prototype presented in Section 4.4 on page 91 for four different non-reactive and reactive xDSLs.

[4] http://codecover.org/documentation/references/javaMeasurement.html

[5] https://www.cambridge.org/resources/052182060X/#programs

[6] https://gitlab.univ-nantes.fr/naomod/faezeh-public/xtdl/-/tree/master/publications-data/SLE22-paper-data

We executed the test cases on models, a total of 360 test cases on 82 models. We observed that our *Coverage Computation* tool successfully computed their coverage and saved the coverage measurements in XMI files. Afterward, we executed the *Mutation Testing* component which resulted in running 360 TDL test cases on 12.951 mutants generated from 82 models, cumulatively. This indeed includes two steps for each model: (1) executing the TDL test suites provided for the model on both the original model and its generated mutants; and (2) reporting the mutation analysis result, such as percentage of the applied mutation operators, number of the generated mutants, mutation score (i. e., percentage of killed mutants), and test execution result for each mutant. The *Mutation Testing* component successfully identified the mutants killed by the related test cases and computed the mutation score of each considered test suite. In total, among 12.951 mutants, 10.678 of them were killed by our 360 written test cases (82.45 %). One of the main reasons for not reaching a 100 % mutation score was that some of the mutants had equivalent behavior to the original model, so the test suites were not able to kill them.

Answering RQ2. Given an xDSL, the initial coverage measurements of its conforming models (i. e., computed based on the execution traces) may not be as detailed as needed by the domain experts. In the second research question, we are questioning the level of required customization for each xDSL to have the intended coverage measurements for their conforming models. For xMiniJava, xArduino, and xPSSM their operational semantics do not provide enough information about the models' executions required for realizing the intended coverage measurements. To overcome this, we used the *DSL-Specific Coverage Rules* metalanguage for defining coverage rules, and in total, we have implemented 25 coverage rules of different types in 151 LoC. Therefore, using a few LoC, we efficiently realized the intended coverage computation for our xDSLs.

Answering RQ3. The third research question targets the validity of our proposed *xModel Coverage Computation* component. To answer it, we compared the coverage matrix generated by our proposed component for the xMiniJava tests with that of generated by CodeCover for the statement coverage of equivalent JUnit tests. For example, Table 4.3 lists the coverage percentage for five randomly selected tests calculated by each tool. With our approach, it is calculated by dividing the number of covered model elements by the total number of traceable elements while with CodeCover is the percentage of covered Java statements. The slight differences between the results are because of some additional lines of code that CodeCover considers while they are not a statement (e. g., the closing curly brace of the `if` statements). We manually verified that the coverage status of each Java statement by each JUnit test is the same for its equivalent xMiniJava element by its related

Test Cases	Our Coverage	CodeCover Coverage
test 1	23/33 = 69.70%	24/35 = 68.57%
test 2	7/36 = 19.44%	7/43 = 16.28%
test 3	28/49 = 57.14%	31/56 = 55.36%
test 4	51/54 = 94.44%	55/60 = 91.67%
test 5	46/119 = 38.66%	57/144 = 39.58%

Table 4.3: Coverage for a set of tests calculated by our approach and CodeCover

test, meaning that our approach provides the same result for the end user. This result shows the validity of our approach.

4.5.3 Threats to Validity

Similarly to the threats to the validity of our evaluation in Section 3.8.4 on page 77, here we also have an external threat that the proposed quality measurement facilities might not work as expected for other modeling languages or in other language workbenches. In addition, we have used the DSL-specific coverage rules metalanguage for three xDSLs, hence there is a threat that it may not be adequate for defining coverage rules for other xDSLs. Also, the validity of our coverage computation approach is compared with one existing coverage tool. As there exist other tools like JaCoCo [76] and Cobertura [38], we can further support the validity of our approach in the future by comparing it with other existing tools.

4.6 Conclusion

In this chapter, we added test quality measurement support to our proposed testing framework by introducing a generic coverage computation approach for a wide range of xDSLs as well as integrating our tool with a generic model mutation tool (i. e., WODEL). Given an xDSL, (i) we compute the coverage of the tests for a given model by analyzing the xDSL definition, the execution trace of the tested model, and the xDSL-specific coverage rules (if available); and (ii) we can compute the mutation score of the tests for a given model if the mutation operators are provided for the xDSL. In conclusion, we observed that an automated and customizable approach for test quality measurement enriches the DSL definition with further V&V techniques at a reasonable cost. More precisely, a language engineer just provides the coverage rules (it is even optional) and the mutation operators for his/her xDSL, and then the test quality measurement facilities are enabled for all of its conforming models.

Discussions and Improvements. Quality measurement techniques are the enabler of many V&V techniques such as test case generation, improvement, minimization, selection, and prioritization, among others. Accordingly, in chapter 6 on page 129, we introduce several interesting lines of research that can be followed in the future.

In addition to that, we have identified some limitations in the contributions of this chapter. Our proposed coverage computation approach allows language engineers to define DSL-specific coverage rules, but it does not currently detect potential conflicts between the rules. For example, the metalanguage must prohibit the definition of inclusion and exclusion rules for the same context metaclass. Conflict detection is also essential for the application of coverage rules because they are currently applied in order repeatedly until a fixed point is reached i. e., until the coverage matrix becomes steady. This means in case of having rules with conflicts, it is probable that the execution enters into an infinite loop.

Although test quality computation helps to measure the strength of the written test cases, there are still other concerns that must be addressed. When test cases fail on a model, it means the model has a defect that must be first localized and then fixed. After debugging, all test cases pass on the model, but if they are not strong enough (based on the result of the quality measurement), test improvement is needed. In the next chapter, we discuss these concerns.

Chapter 5

Test Case Debugging and Improvement

5.1 Introduction

A failed test case is an *alert* for the domain expert that tells there is a defect in the model under test causing the failure. For trivial test cases, the test report may provide adequate information about the cause of failure but localizing faults can be more difficult for more complex test cases. Therefore, there is a need for fault localization techniques to help the domain expert in finding the faults. In addition, although by testing and debugging a model we can ensure the correctness of its current version, there is always a threat of regression faults, meaning that the model may be affected by faults in future updates. With mutation analysis, we can measure the strength of the existing test suites in detecting potential faults, and so checking if an improvement is required. However, improving a test suite is a complicated task for the domain expert.

To meet these needs, there are several well-known techniques in the context of software testing. For fault localization, there are both manual and automatic techniques, such as *interactive debugging* and *Spectrum-Based Fault Localization (SBFL)*, respectively [150]. The former allows the tester to perform a step-by-step observation of the System Under Test (SUT) behavior as triggered by the test case. The latter provides a suspiciousness-based ranking of the SUT's components (e. g., statements of a Java program) using the results of test cases and their corresponding coverage information. Also, *test amplification* techniques have recently emerged, that aim at automatically improving existing manually-written test suites towards a specific goal e. g., to increase the accuracy of fault detection [42]. However, to the best of our knowledge, they are not yet adapted for the context of model testing.

Leveraging these techniques in the context of model testing faces some challenges.

For manual fault localization using interactive debugging, while there are interesting debugging approaches for models [30, 37], none is able to support the step-by-step execution of a failed test case *along with* the step-by-step execution of the tested model to help the domain expert in localizing the fault causing the test failure. For adapting SBFL, the barrier roots in the limitations of existing testing frameworks and coverage computation approaches in the context of xDSLs as already mentioned in Sections 3.1 and 4.1 on pages 47 and 79, respectively. Lastly, the test amplification techniques are mainly developed for specific programming languages (e. g., Java [43], Pharo Smalltalk [1], and Python [126]) and they are not yet studied for the context of xDSLs.

On the foundation of our proposed facilities for test case definition, execution (Chapter 3), and evaluation (Chapter 4), this chapter offers interactive debugging, SBFL fault localization, and test amplification techniques generically in the context of xDSLs. The interactive debugging facility coordinates the initialization and the online interplay of two debugger instances to debug a TDL test case along with its model under test. The SBFL facility reuses an existing collection of SBFL techniques [142] and calculates the suspiciousness-based ranking of the models' elements. The test amplification facility automatically improves a given TDL test suite of a model in detecting potential regression faults. In the following, we first introduce an overview of our proposed facilities in Section 5.2. We then provide more details about each in the subsequent sections: interactive debugging in Section 5.3 on page 102, fault localization in Section 5.4 on page 105, and test amplification in Section 5.5 on page 106.

5.2 Overview

Figure 5.1 shows an overview of our proposed test debugging and amplification facilities. Using the *Interactive Debugging* component (label 1), the domain expert can debug interactively the test case and its model under test at the same time, hence observing gradually the model's reaction to the reception of requests from the test case. To this end, this component controls both the test case execution by the *TDL Interpreter* (i. e., already proposed in Section 3.5 on page 62) and the model execution by the *Execution Engine* (at the bottom left). More details are provided in Section 5.3 on page 102.

Through the *Fault Localization UI* component (label 2), the domain expert can request for running the *Fault Localization* component (label 3). It provides an automatic debugging approach by applying a set of collected Spectrum-Based Fault Localization (SBFL) techniques taken from [142]. It reads the test results produced by the *TDL Interpreter* and the coverage matrix constructed by the *Coverage Computation* component (i. e., already proposed in Section 4.2 on page 80)

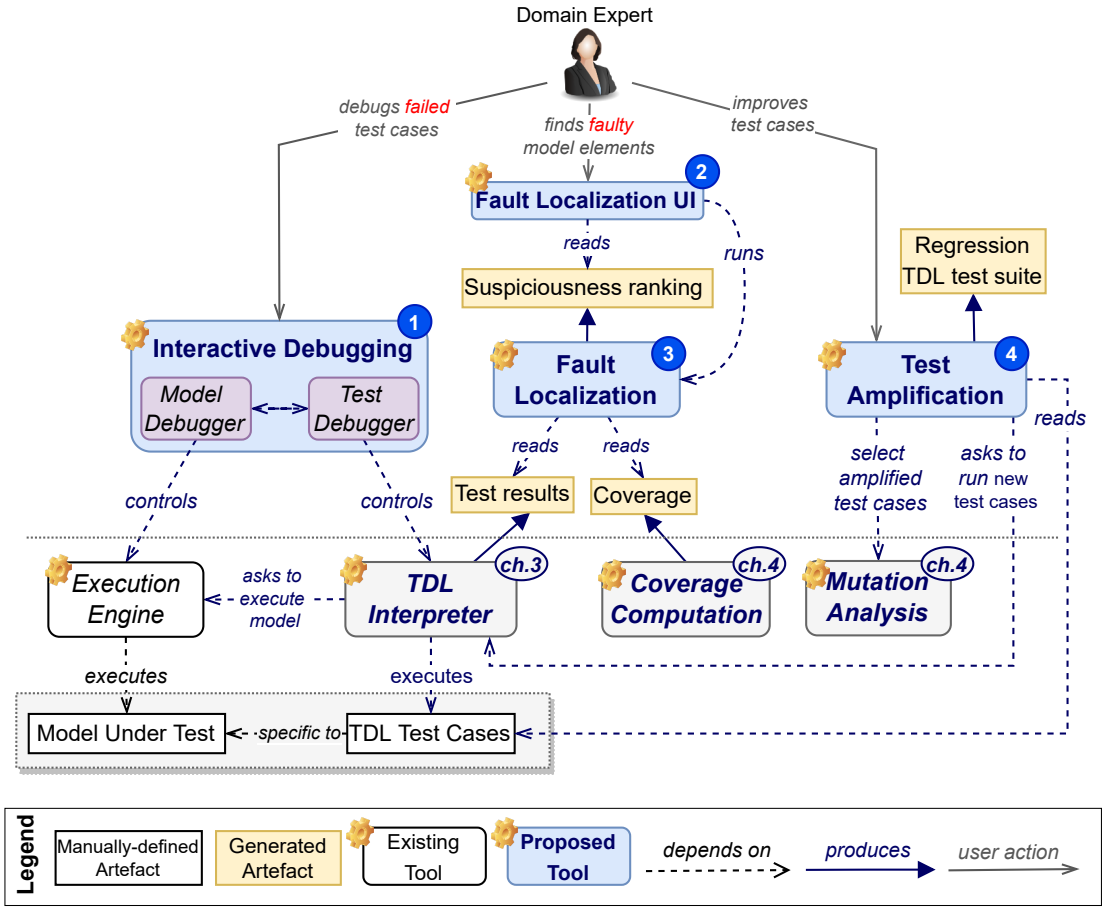


Figure 5.1: An overview of the test debugging and amplification facilities

to generate the suspiciousness-based ranking of the model's elements. Such ranking helps in debugging the model as it directly positions the location of the faults. In Section 5.4 on page 105, more details are given.

Finally, to improve the TDL test cases (i. e., written by the domain expert) in efficiently detecting potential regression faults, we propose an automated approach for amplifying TDL test suites of executable models. As shown in Figure 5.1, the *Test Amplification* component (label 4) generates a regression TDL test suite for the model under test which is an improved version of the initial TDL test suite regarding finding regression faults. This component relies on the *TDL Interpreter* for the execution of new test cases and on the *Mutation Analysis* component (i. e., already proposed in Section 4.3 on page 89) for selecting those new test cases improving the initial mutation score. In Section 5.5 on page 106, we explain the details of our test amplification approach.

5.3 Manual Debugging of Models' Tests

In the context of software testing, most of the popular testing frameworks (e. g., JUnit) are compatible with interactive debugging facilities (e. g., `jdb`[1]). Among other possibilities, this allows the tester to perform a step-by-step observation of the SUT behavior as triggered by the test case. But for this to work, it must be possible to execute step-by-step not only the SUT but also the test case itself. In other words, it must be possible to perform interactive debugging for both the test case and the SUT *in unison*. When the test case and its SUT are both implemented using the same language (such as Java programs and their JUnit test cases), this is trivial to achieve by using a single debugger instance, since both the SUT and the test case are then executed as one single executable program. However, in the context of this thesis, the test case and the SUT are two different executable models conforming to two different languages. This means debugging models' test cases needs:

1. being able to debug an executable model itself.
2. initializing two debugger instances at the same time, one for the test case and another for the model under test, while making sure the debugging services remain consistent when used in two different debuggers, and coordinating the communication between the two debuggers.

To the best of our knowledge, the first matter is already addressed for both EMF [29, 30] and UML [37] models, and we use the interactive debugging approach proposed by Bousse et al. [30] as it supports the xDSLs considered in the context of this thesis. More specifically, their approach can be configured for a specific xDSL and then can be used to debug its conforming executable models. However, the second challenge is still open and the remainder of this section explains our proposal to resolve it.

5.3.1 Adapting Interactive Debugging for TDL

When running a TDL test case with an interactive debugger, as soon as the execution reaches a point where the test case makes a request to the model under test (e. g., a TDL message sending an event to a reactive model under test), one can expect to be able to "jump" from the debugger of the TDL test case to the debugger of the model under test, and to switch to observing the model's behavior. More precisely, this can be expected when the modeler either sets a breakpoint inside the model under test or wishes to *step into* the processing of the request sent to the model by the test case (e. g., an event). To meet these expectations

[1] https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html

in our approach, we make the following minor adaptations to common interactive debugging services (the one previously introduced in Section 2.3.5.1 on page 36) for debugging TDL test cases of executable models:

- The **resume** operator: It continues the test case execution until a breakpoint is reached either in the test case or in the model under test.
- The **step over** operator: It continues the execution until the end of the current step or until a breakpoint is reached in the test case or in the model under test.
- The **step into** operator: It continues the execution until either some inner step is reached (if any) or the current step is ending. If in the current step, the test case sends a request to the model under test, the *step into* operator pauses the execution inside the model under test at the very beginning of processing the sent request.

For example, Figure 5.2 illustrates an interactive debugging scenario for the running example using our redefined debugging services. Here we see a situation where the modeler has set a breakpoint (shown as a filled colored circle) in the faulty TDL test case (previously shown in Figure 3.2(**b**) on page 52), on the TDL message that sends an IRSensor_detected event for the infrared sensor to the xArduino model under test. When the test case execution reaches this TDL message, it pauses because of the breakpoint. The modeler may wish to investigate how this event will be processed in the xArduino model. So by using the *step into* operator, the execution pauses at the beginning of processing said event by the xArduino model i. e., the if infrared sensor == 1 condition (label 1).

As the if condition is satisfied, the execution enters into its body. Thereafter using the *step over* operator in the model debugger, the modeler can observe first the buzzer turns on by executing the `buzzer = 1` ModuleAssignment (label 2) and then it turns off due to executing the `buzzer = 0` ModuleAssignment (label 3). By using the *step over* operator once again, the faulty `buzzer = 0` ModuleAssignment will be executed (label 4). At this point, the modeler observes that instead of setting the buzzer to 1 (i. e., turning on the buzzer for the second time), it is again set to 0, hence discovering the defect in the value of the ModuleAssignment. From now on, the domain expert can use either the *step over* operator to continue with step-by-step model debugging or the *resume* operator. In both cases, when the model execution reaches the end of the if statement, the test case debugger resumes, so the next TDL message can be executed (i. e., the TDL message after the breakpoint).

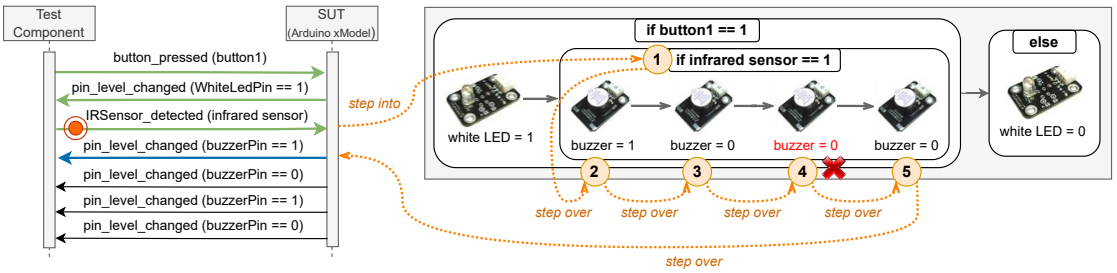


Figure 5.2: A sample scenario of performing interactive debugging for the running example

5.3.2 Initialization and Coordination of Two Interactive Debuggers

As previously mentioned, debugging a TDL test case requires two interactive debugger instances, one for the test case and one for its model under test. This section explains how we spawn and coordinate them using a sample scenario shown in Figure 5.3. The domain expert starts the process by requesting to debug a TDL test suite containing at least one TDL test case. This results in initializing a debugger for the test suite, preparing the TDL Interpreter, and pausing the execution where reaching the first breakpoint (if any). In the scenario of Figure 5.3, it pauses at the very beginning of the test suite execution as we configured a breakpoint there. Then, the domain expert can use the *step over* service of the debugger to start the execution of the first test case.

As described in Section 3.5 on page 62, for test case execution, the TDL Interpreter first activates the test configuration of the test case. In general, two situations may happen:

- the test case is for a *non-reactive* model: an instance of a model execution engine must be configured.
- the test case is for a *reactive* model: an instance of an event manager must be configured.

Hereupon, the internal behavior of the test case can be executed step-by-step using the services of the test case debugger, such as *step into*. It is also possible to put a breakpoint on the test case and *resume* the execution.

When the test component requests an execution in the model under test, if the domain expert wants to observe the model's behavior upon receiving that request, a second debugger is required to be initialized for the model. To do this, we added new functionalities to our *TDL Interpreter* component. As shown in Figure 5.3, the first time the domain expert chooses the *step into* operator (in the test case debugger) when the test component sends a request to the model (e. g., requesting

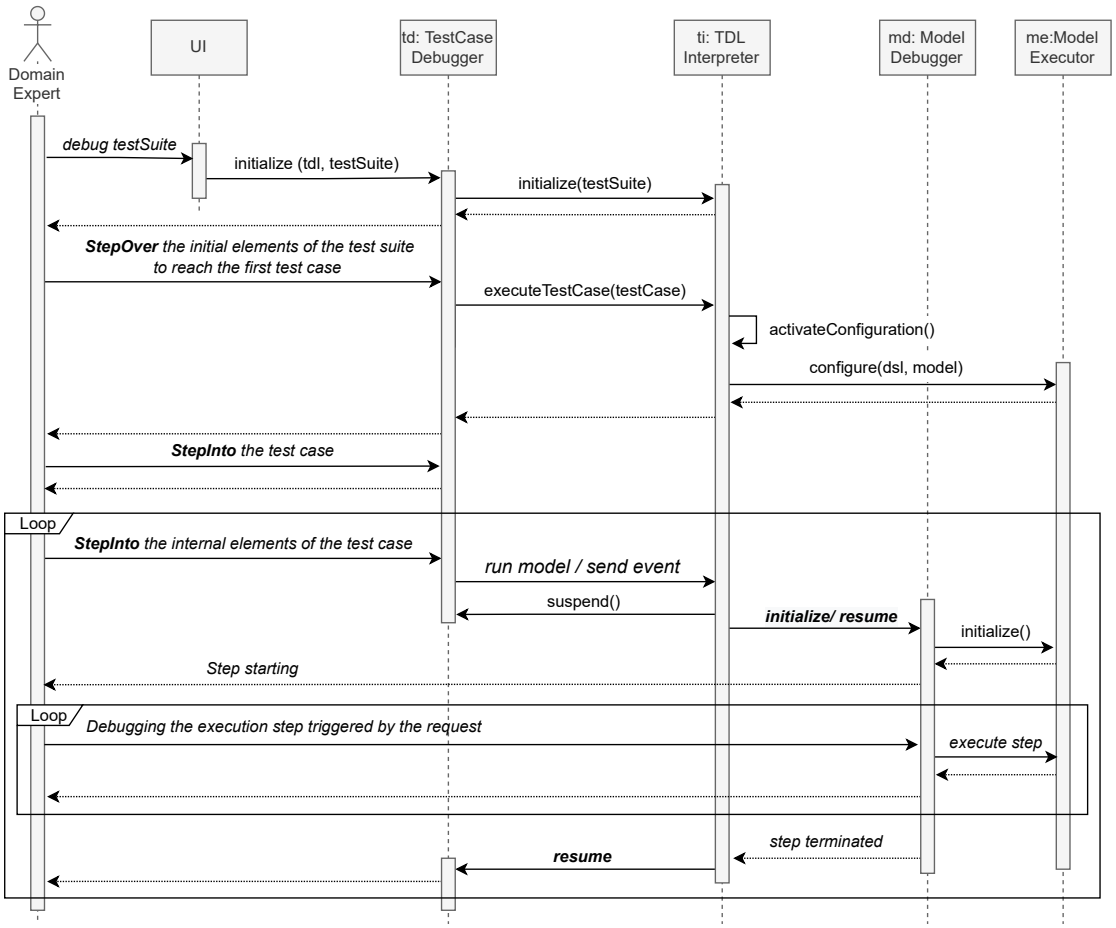


Figure 5.3: One possible interactive debugging scenario for a TDL test case written for an executable model

model execution or sending an accepted event), the TDL Interpreter initializes a debugger for the model. Hereafter, the TDL Interpreter pauses and resumes the model debugger according to the debugging services chosen by the domain expert in the test case debugger, based on their redefined semantics presented in Section 5.3.1 on page 102. It also deactivates the test case debugger for the active time of the model debugger to ensure their consistency.

5.4 Automatic Debugging of Models' Tests

Manual debugging is cumbersome for test cases of large and/or complex models, thus there is a need for advanced techniques which can help in finding the models' faults. This section proposes an automatic model debugging approach based on the

Spectrum-Based Fault Localization (SBFL) techniques [150]. To apply SBFL for finding faults in a model, we need the execution result and the coverage information of the model's test cases (as already described in Section 2.3.5.2 on page 37). Using our proposed generic testing framework, we can define test cases for any executable model and the framework automatically produces both the test execution result and the test coverage measurements—described earlier in Sections 3.6 and 4.2 on pages 71 and 80, respectively. Therefore, it enables us to also offer SBFL for the context of xDSLs.

As already mentioned in Section 2.3.5.2 on page 37, SBFL is usually applied at the statement level for software programs, meaning that it uses the statement coverage of a program and calculates the suspiciousness of each statement [150]. In this thesis, we adapt SBFL for the context of xDSLs by substituting the notion of statement with the more generic concept of *model element*, resulting in redefining SBFL parameters as follows:

- N_{CF}: number of failed test cases that cover the model element
- N_{UF}: number of failed test cases that do not cover the model element
- N_{CS}: number of successful test cases that cover the model element
- N_{US}: number of successful test cases that do not cover the model element
- N_C: total number of test cases that cover the model element
- N_U: total number of test cases that do not cover the model element
- N_S: total number of successful test cases
- N_F: total number of failed test cases

The resulting *Fault Localization* component of our testing framework is shown at the center of Figure 5.1 on page 101. Considering a TDL test suite of a model, this component uses the test execution result generated by the *TDL Interpreter* and the coverage matrix produced by the *Coverage Computation* component to calculate the suspiciousness-based ranking of the model's elements using SBFL techniques. Currently, we support 18 existing SBFL formulas shown in Table 5.1 which have been collected by Troya et al. [142] by investigating primary studies proposing concrete SBFL techniques.

5.5 Test Amplification for Executable Models

In this section, we propose a generic, automated approach for amplifying test suites of executable models for regression testing. Our approach only supports *reactive* xDSLs as explained below.

Table 5.1: Supported SBFL formulas (taken from [142])

Technique	**Formula**
Arithmetic Mean [154]	$\frac{2(N_{CF} \times N_{US} - N_{UF} \times N_{CS})}{(N_{CF} + N_{CS}) \times (N_{US} + N_{UF}) + (N_{CF} + N_{UF}) \times (N_{CS} + N_{US})}$
Barinel [2]	$1 - \frac{N_{CS}}{N_{CS} + N_{CF}}$
Baroni-Urbani & Buser [149]	$\frac{\sqrt{N_{CF} \times N_{US}} + N_{CF}}{\sqrt{N_{CF} \times N_{US}} + N_{CF} + N_{CS} + N_{UF}}$
Braun-Banquet [150]	$\frac{N_{CF}}{max(N_{CF} + N_{CS}, N_{CF} + N_{UF})}$
Cohen [110]	$\frac{2 \times (N_{CF} \times N_{US} - N_{UF} \times N_{CS})}{(N_{CF} + N_{CS}) \times (N_{US} + N_{CS}) + (N_{CF} + N_{UF}) \times (N_{UF} + N_{US})}$
DStrar [148]	$\frac{(N_{CF})^*}{N_{CS} + N_F + N_{CF}}$
Kulczynski2 [110]	$\frac{1}{2} \times (\frac{N_{CF}}{N_{CF} + N_{UF}} + \frac{N_{CF}}{N_{CF} + N_{CS}})$
Mountford [149]	$\frac{N_{CF}}{0.5 \times ((N_{CF} \times N_{CS}) + (N_{CF} \times N_{UF})) + (N_{CS} \times N_{UF})}$
Ochiai [3]	$\frac{N_{CF}}{\sqrt{N_F \times (N_{CF} + N_{CS})}}$
Ochiai2 [19]	$\frac{N_{CF} \times N_{US}}{\sqrt{(N_{CF} + N_{CS}) \times (N_{US} + N_{UF}) \times (N_{CF} + N_{UF}) \times (N_{CS} + N_{US})}}$
Op2 [110]	$N_{CF} - \frac{N_{CS}}{N_S + 1}$
Phi [100]	$\frac{N_{CF} \times N_{US} - N_{UF} \times N_{CS}}{\sqrt{(N_{CF} + N_{CS}) \times (N_{CF} + N_{UF}) \times (N_{CS} + N_{US}) \times (N_{UF} + N_{US})}}$
Pierce [150]	$\frac{(N_{CF} \times N_{UF}) + (N_{UF} \times N_{CS})}{(N_{CF} \times N_{UF}) + (2 \times N_{UF} \times N_{US}) + (N_{CS} \times N_{US})}$
Rogers & Tanimoto [99]	$\frac{N_{CF} + N_{US}}{N_{CF} + N_{US} + 2(N_{UF} + N_{CS})}$
Russel-Rao [120]	$\frac{N_{CF}}{N_{CF} + N_{UF} + N_{CS} + N_{US}}$
Simple matching [150]	$\frac{N_{CF} + N_{US}}{N_{CF} + N_{CS} + N_{US} + N_{UF}}$
Tarantula [81]	$\frac{\frac{N_{CF}}{N_F}}{\frac{N_{CF}}{N_F} + \frac{N_{CS}}{N_S}}$
Zoltar [77]	$\frac{N_{CF}}{N_{CF} + N_{UF} + N_{CS} + \frac{10000 \times N_{UF} \times N_{CS}}{N_{CF}}}$

5.5.1 Scope

As already mentioned in Section 2.3.6 on page 39, there are three main steps to amplify test cases for regression testing:

- creating new test cases by modifying the test input data of existing test cases.
- executing the new test cases on the system under test and capturing the system's reaction to the new data through its execution trace.
- generating test oracles for the new test cases by analyzing the execution trace.

Therefore, to amplify the test cases of a given model, we need a clear definition of test data—both test input data and expected output (i. e., for generating test oracles—as well as the model's execution trace.

In the context of xDSLs, test data definition differs between non-reactive and reactive xDSLs. For a non-reactive xDSL, it relies on the definition of xDSL's runtime state e. g., in the TDL test case of Listing 3.8 on page 63, `buttonPin (level = '1')` is the test input data and `project (pinChanges = whiteLedPin (level = '1'), ...)` is the expected output. However, for a reactive xDSL, it is based on the definition of xDSL's behavioral interface e. g., in the TDL test case of Listing 3.9 on page 64, `button_pressed (button1)` is the test input data and `pin_level_changed (whiteLedPin (level = '1'))` is the expected output.

We also already mentioned in Section 2.2.4 on page 21 that an execution trace of an executable model captures several different pieces of information, one of which is the set of changes in the model's runtime state during the execution. According to the definitions given in Section 2.2.3 on page 13, the runtime state definition merges with the abstract syntax resulting in an execution metamodel. This means that part of the execution trace related to the changes in the model's runtime state comprises instances of the execution metamodel that each can also be seen as a graph (the so-called *type graph* as introduced in [25]). However, an execution trace of a reactive model also captures all the occurrences of the exposed events (i. e., the event occurrences exposed by a running reactive model) as a list.

Therefore, for generating oracles from the execution trace of a non-reactive model, we should perform a graph analysis which is a complex task; as explained above, the oracles must be defined based on the changes of the model runtime state which are captured in the model execution trace as instances of the execution metamodel so as graphs. While for reactive models, we can just simply analyze the list of the exposed events occurrences. In this section, we introduce a test amplification approach for reactive xDSLs, while supporting non-reactive ones is left to future work.

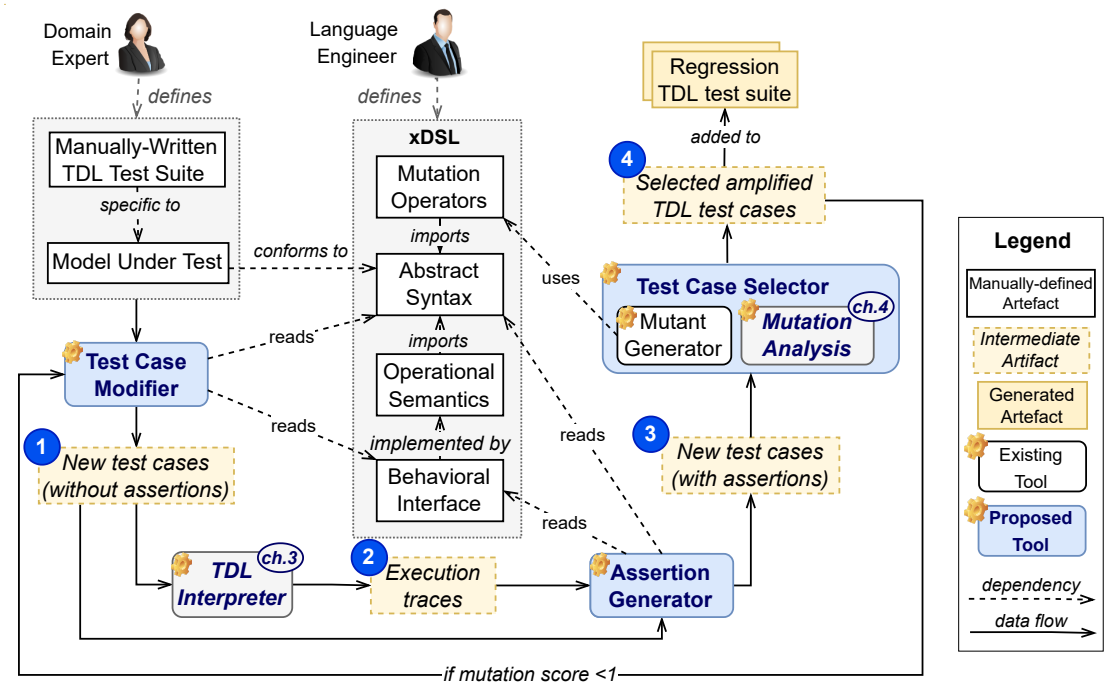


Figure 5.4: An overview of the test amplification approach

5.5.2 Approach Overview

Figure 5.4 shows an overview of our proposed approach with the same roles as the previous chapters (e. g., Figure 3.1 on page 49). First, a language engineer (on the top center) who defines a reactive xDSL according to the definitions given in Section 2.2 on page 10 and a set of mutation operators for the xDSL. Second, a domain expert (on the top left) who defines a model (using the xDSL) and a TDL test suite for them.

The first component of the approach is the *Test Case Modifier* (on the left side). It takes the manually-written test suite and the model under test, and generates new test cases by modifying the given test suite (label 1). This modification involves changing the test input data and removing the existing assertions since they are no longer valid due to the data changes. As the test input data are specific to the xDSL that the tested model conforms to, this component uses the xDSL definition for performing the modification. The new test cases are then given to the *TDL Interpreter* (on the bottom left) (i. e., already proposed in Section 3.5 on page 62) to execute the test cases on the model and produce their execution traces (label 2).

Our second component is called *Assertion Generator* (on the bottom right) and follows the idea of regression oracle checking [153]. In this technique, assertions are generated based on the execution traces to improve the strength of the regression

testing. Accordingly, after running each new test case, our proposed component analyzes the execution trace of the model under test to generate assertions for the test case based on the model's reaction to the new test input data. Again, as the execution trace comprises data conforming to the xDSL that the model conforms to, this component also uses the abstract syntax and the behavioral interface of the xDSL definition, this time for generating assertions. At the end, its output is a set of new test cases with assertions for regression testing (label 3).

The *Test Case Selector* is the third component of the approach (on the right side). It is in charge of filtering the generated test cases based on some given criteria. Currently, we consider the ability of new test cases in improving the fault localization capability, so this component uses the *Mutation Analysis* facility already presented in Section 4.3 on page 89. Accordingly, it uses the WODEL mutant generator which produces mutants out of the model under test if the language engineer provides a set of mutation operators for the xDSL (on the top center) [62]. It then runs the new test cases on the mutants using the *Mutation Testing* component and keeps those improving the initial mutation score (i. e., the mutation score of the given manually-written TDL test suite). The process can be iterated on the selected new test cases based on some stop criteria such as reaching a 100 % mutation score (label 4).

5.5.3 Test Case Modification

The first step of our test amplification process is the modification of existing test cases. This involves performing two tasks on each test case of the considered test suite: modifying the test input data and removing the assertions. The former aims at putting the model under test in unexplored runtime states, and the latter is required since changing the input data makes the existing assertions invalid.

For the former task, we call *modifier* an operator that, when applied to a specific element of an existing test case, generates a new test case that is identical to the former one but for a single modification. A modifier can be applied multiple times on the same test case, yielding a different result depending on the chosen element of the test case. As some modifiers may produce too many different new test cases from a single test case, each modifier may possess its own *application policy*, which tells on which elements and how many times the modifier will be applied on each test case. In the proposed approach, we use the following sets of modifiers.

5.5.3.1 Modification of primitive data

We adapt two existing sets of modifiers for modifying test input data that comprise primitive values: (1) the operators proposed by Danglot et al. in [43] for the amplification of JUnit test cases, and (2) the modifiers used by the Pitest mutation

testing tool [39] for putting Java programs in new runtime states. The resulting modifiers are as follows:

- A *numeric* value n is replaced by either 1, -1, 0, $n + 1$, $n - 1$, $n \times 2$, $n \div 2$, or with another existing value of the same type.
- A *string* value is modified by either adding a random character, removing one of its characters randomly, arbitrarily replacing one of its characters with a random character, or replacing the string with a random string of the same size.
- A *boolean* value is negated.

Each of these modifiers is applied as many times as possible on each considered test case. For instance, the integer modifier will be applied 16 times on a test case containing an event occurrence with two integer parameters (2×8 possibilities).

5.5.3.2 Modification of event sequences

As explained in Section 3.3.2 on page 51, the test input data of a TDL test case written for a reactive model is composed of a sequence of event occurrences. According to the reactive xDSL the tested model conforms to, such occurrences are instances of the accepted events of the xDSL's behavioral interface. Each event occurrence may have a set of parameters pointing to the model elements. Following these considerations, we propose the following modifiers to generate new test cases by modifying the input event sequences of a given test case:

- *Event Duplication*: Duplicate an existing event occurrence. Applied on each possible event occurrence of the test case.
- *Event Deletion*: Removes an event occurrence. Applied on each possible event occurrence of the test case.
- *Event Permutation*: Performs a permutation of two random event occurrences to generate a new test case.
- *Event Creation*: Creates a new event occurrence in the test case. First, the available accepted events of the xDSL's behavioral interface that are not used in the test input data of the given test case are collected. The availability is verified by analyzing whether, for the unused events, a value can be set to their parameters using information from the model under test. If possible, this operator adds new event occurrences to the input event sequence by creating all the possible instances of the available accepted events. In particular, it generates one new test case per event addition as well as a new test case containing all the new instantiated event occurrences. In the latter case,

when several new event occurrences are added, this new input is modified using other operators such as event duplication, event deletion, and event permutation to generate more new test cases.

- *Event Modification*: Analyzes the model under test to find alternative values for the parameters of the events (i. e., other values of the same type). If any are found, the values of the event parameters are replaced with the alternatives. With each possible modification, it generates a new test case.

The output of this step is a set of new test cases but still without any assertion. As mentioned in Section 5.5.1 on page 108, our approach requires execution traces to generate regression assertions. Accordingly, we execute each new test case on the model under test using the *TDL Interpreter* to capture an execution trace (label 2 in Figure 5.4 on page 109).

5.5.4 Assertion Generation

As explained in Section 5.5.1 on page 108, the execution trace of a reactive model comprises a sequence of exposed event instances, according to the behavioral interface of the xDSL that the model conforms to. Since the test cases are implemented in TDL, the assertions must be generated in TDL as well. Thus, the main role of the assertion generator is to transform the exposed event instances to TDL elements. This transformation uses the definition of both the behavioral interface and the abstract syntax of the xDSL because event instances conform to the behavioral interface and may carry EObjects of the model under test, each conforms to the abstract syntax metamodel.

5.5.5 Amplification Example

For example, Figure 5.5 on page 113 demonstrates the correct version of the xArduino sample model (Figure 2.2 on page 12). Thus here the TDL test case of Figure 3.2(**b**) on page 52 passes, as shown in Figure 5.6 on page 113. However, this test case is checking the Sketch part of the xArduino model, except for the `else` part, so considering a regression fault in the `else` part such as Figure 5.7 on page 114, the test case would not be able to find the fault.

We applied our test amplification approach on the initial test case of Figure 5.6 and Figure 5.8 on page 114 illustrates the output generated as follows: In the test case modification phase, the `Event Creation` modifier added a new event, `button_released (button1)`, to the input event sequence. Then, in the assertion generation phase, a new assertion with expected output `pin_level_changed (white LED == 0)` is generated because according to Figure 5.5 on page 113, by

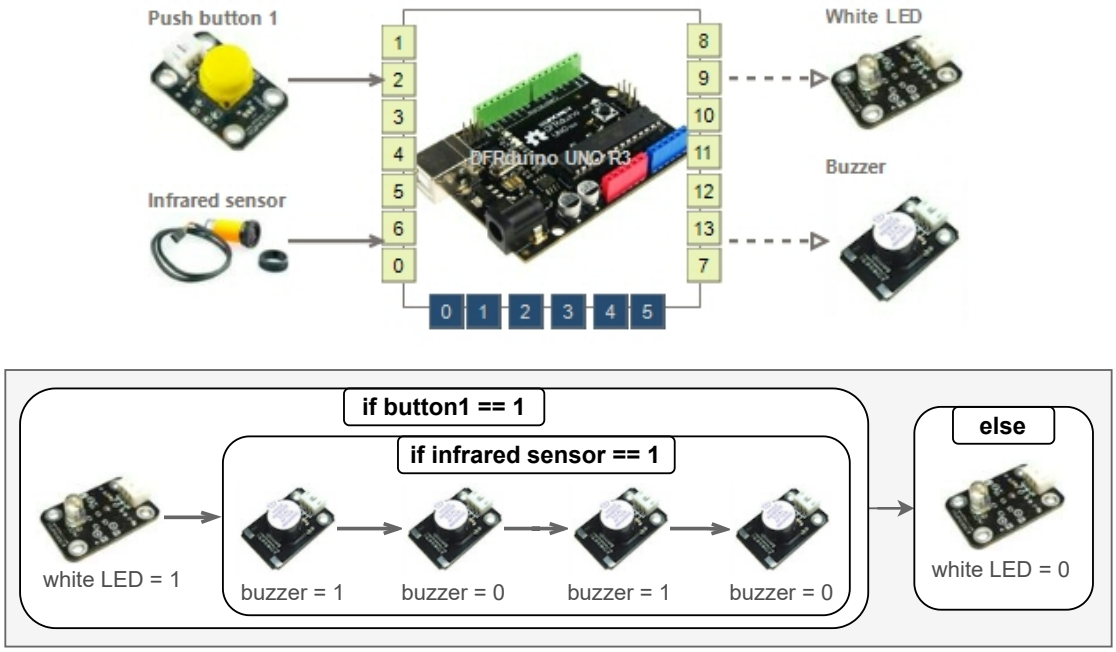


Figure 5.5: The correct version of the xArduino sample model (Figure 2.2 on page 12)

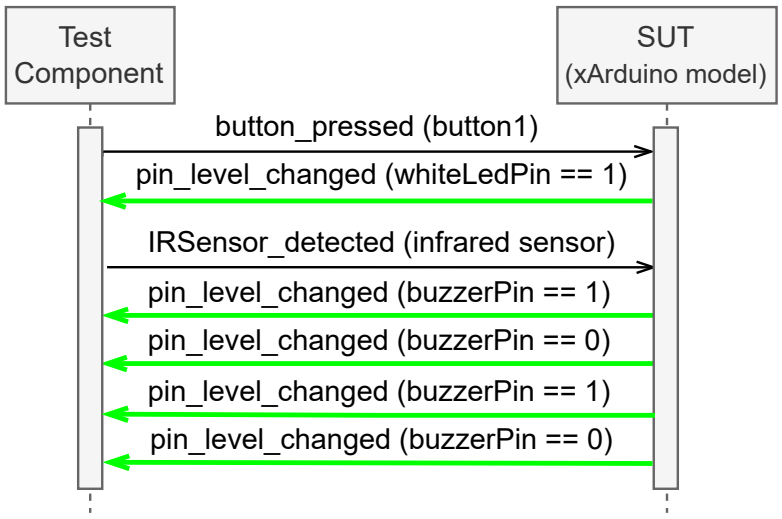


Figure 5.6: The TDL test case of Figure 3.2(**b**) on page 52 that is passed on the xArduino model of Figure 5.5 on page 113

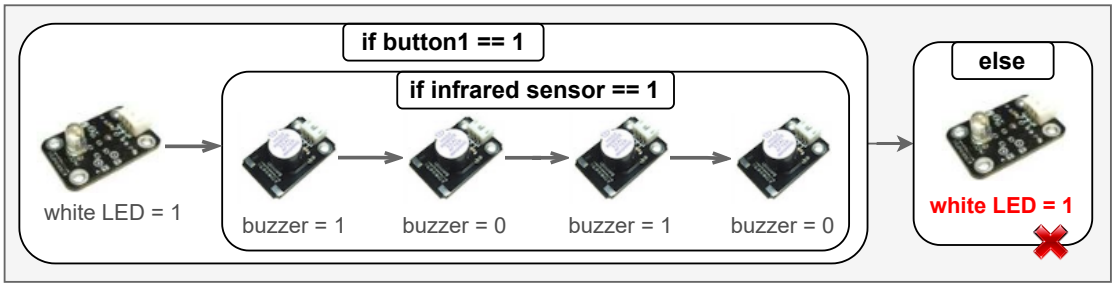


Figure 5.7: A regression fault in the xArduino model of Figure 5.5 on page 113

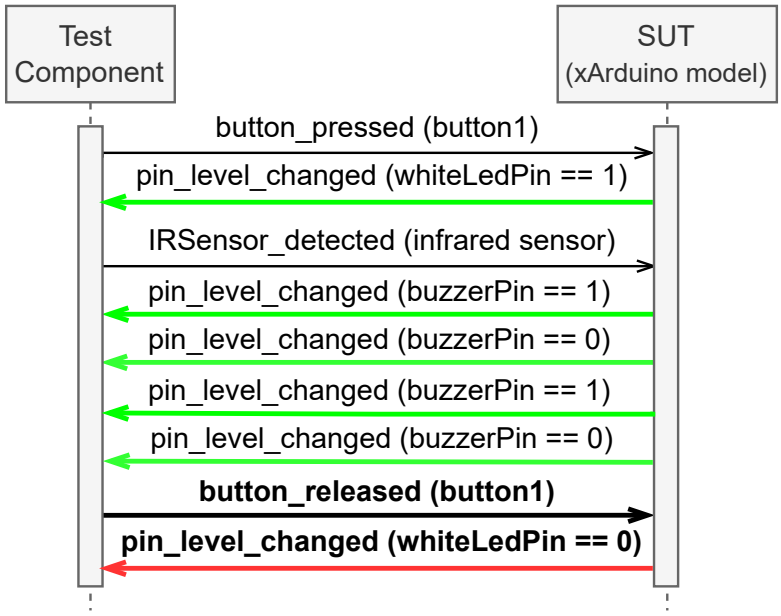


Figure 5.8: An amplified TDL test case generated from the initial test case of Figure 5.6 on page 113 by applying the *event creation* modifier. The last message unsatisfies the assertion then the test case fails, hence finding the regression fault of Figure 5.7

releasing the button the `else` part of the Sketch will be executed which changes the level of the `white LED` to 0.

5.5.6 Test Case Selection

Up to this step, a set of new test cases has been generated, but not necessarily all of them improve the quality of the test suite. In this step, we rely on mutation analysis to evaluate whether a generated test case improves the quality of the test suite. Figure 5.9 on page 115 shows the test case selection process. First, the

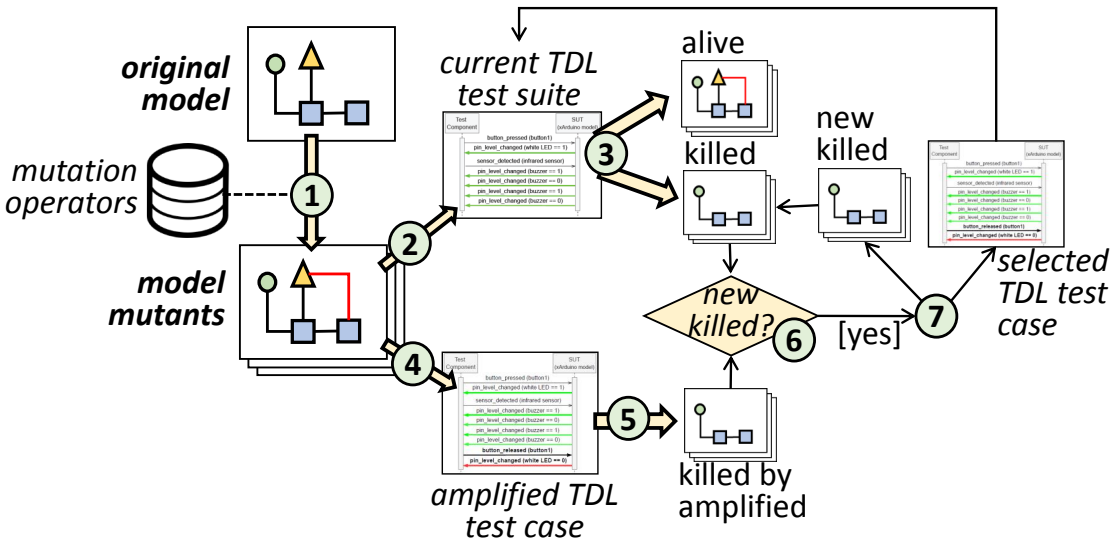


Figure 5.9: Selecting test cases using mutation analysis

original model under test is mutated using a set of mutation operators for the xDSL (step 1). Then, the original test suite is evaluated on the mutants (step 2) to yield a mutation score (step 3). We also keep track of the set of killed and alive mutants.

Next, the process evaluates each amplified test case on the live mutants (step 4) to check whether it fails on any of them (i. e., kills some mutant, step 5) or succeeds (i. e., does not distinguish the mutant from the original model). The amplified test case is selected if it kills at least one live mutant. In such a case, the process incorporates the new killed mutants to the existing set of killed mutants, increasing the mutation score (step 6), and iterates the process on the selected amplified test cases (step 7). The process finishes based on some given stop criteria. Currently, we iterate up to three times while the mutation score is less than 100 %.

For example, considering the xArduino model of Figure 5.7 on page 114 as a mutant of Figure 5.5 on page 113, the initial test case of Figure 5.6 on page 113 passes on the mutant but the new test case of Figure 5.8 on page 114 fails on it. This means that the new test case is able to detect the fault, and therefore that the mutant has been killed, which increases the mutation score. Hence, the test case is added to the regression TDL test suite of the xArduino model.

5.6 Tool Support

This section presents our tool support for debugging (Section 5.6.1 on page 116) and improving (Section 5.6.2 on page 118) TDL test cases.

5.6.1 Debugging Tool

We have implemented both debugging approaches as part of the GEMOC Studio [29]. The *Interactive Debugging* component uses the model debugging framework of the GEMOC Studio [29, 30] for the initialization of two debugger instances, and the Eclipse debug platform[2] for managing their communication and synchronization, all implemented in Java.

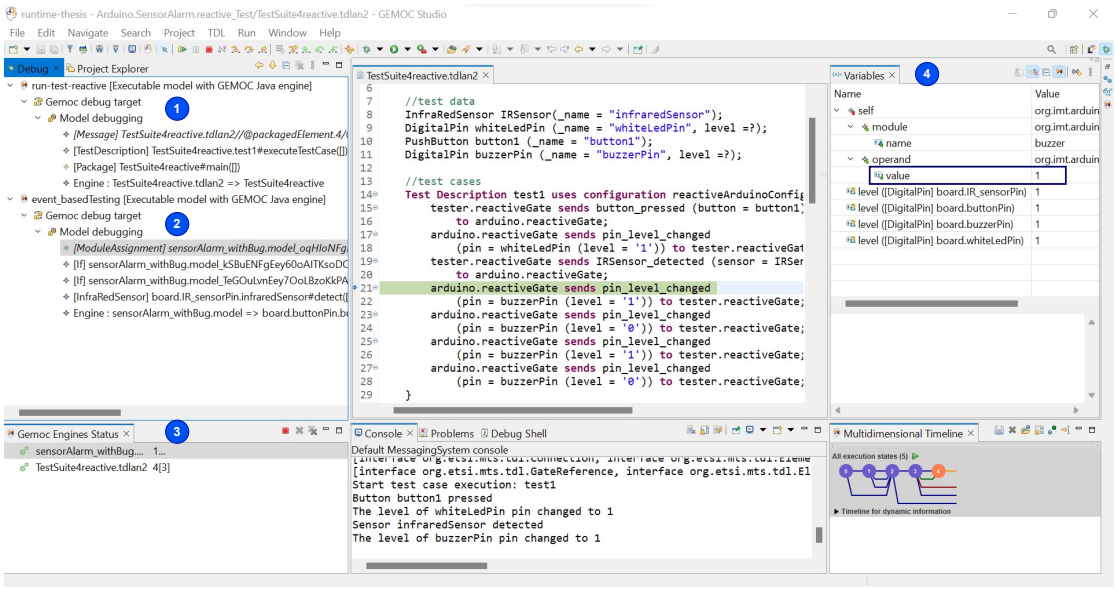
The suspiciousness computation in the *Fault Localization* component is based on that of provided by Troya et al. [142] for fault localization in model transformations, now adapted for general model elements. Currently, our tool supports 18 SBFL techniques but adding new ones is possible. Indeed within the literature, there are approximately 30 SBFL techniques [110, 131, 150]. They all use the set of values explained in Section 5.4 on page 105 (i. e., N_{CF}, N_{UF}, N_{CS}, N_{US}, N_C, N_U, N_S, N_F) to compute the suspiciousness-based ranking. Accordingly, any existing formula defined using the aforementioned variables can be added to the tool.

Figure 5.10 displays two screenshots of the provided test debugging facilities running in the GEMOC studio modeling workbench for the running example. The (**a**) part shows the usage of the interactive debugging facility for the running example. It displays two debugger instances, one for the TDL test suite (label 1) and another for the xArduino model under test (label 2), both running using GEMOC execution engines (label 3). Running the test suite in debug mode, we chose the *step into* operator of the test case debugger where the test case wanted to send an IRSensor_detected event for the infrared sensor to the xArduino model. This paused the test case debugger on the first of the next TDL Message (i. e.,, asserting a pin_level_changed event for the buzzerPin (at the center, line 21)) and enabled a debugger for the model under test. Using the stepping operators of the model debugger (label 2), we observed when the xArduino sample model has received the said event from the test case, the condition of the second if statement has been satisfied and the execution entered into its body. We then identified that the value of the third ModuleAssignment is mistakenly 1 instead of 0 (label 4).

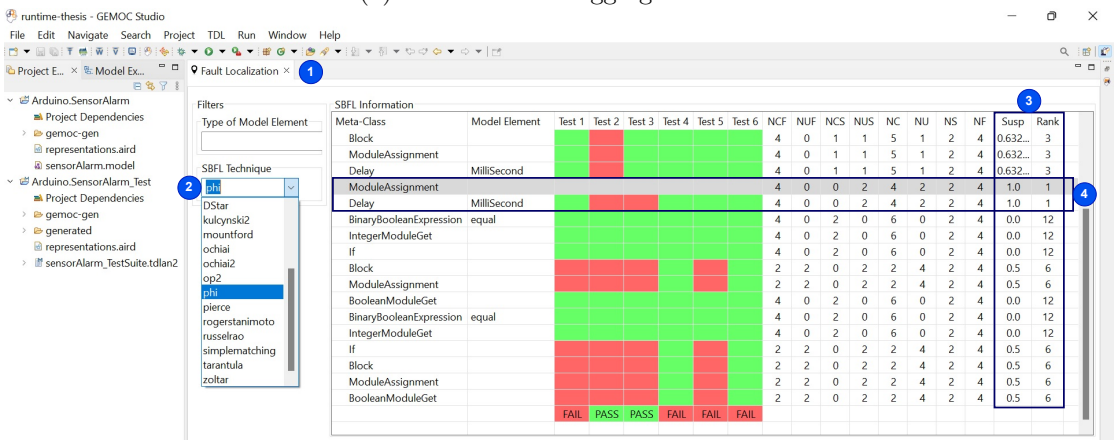
Figure 5.10(**b**) shows a screenshot of our SBFL tool. The source code is available on a public GitLab instance[3]. To run SBFL on a tested model, we provided a graphical view titled "fault localization" (label 1). It lists the traceable elements of the tested model, their coverage status by each test case, the test execution result (at the last row), and the required values for calculating the suspiciousness-based ranking. The view has a drop-down list of the 18 supported SBFL techniques (Label 2). When a technique is selected, the tool calculates the suspiciousness score and the rank for all the model elements and shows the results in the last

[2] https://www.ibm.com/docs/da/rsar/9.5?topic=reference-orgeclipsedebugcore

[3] https://gitlab.univ-nantes.fr/naomod/faezeh-public/xtdl/-/tree/master/fault_localization



(a) Interactive debugging facilities



(b) SBFL facilities

Figure 5.10: Screenshots of the provided test debugging tools running in the GEMOC studio modeling workbench for the running example

two columns (label 3). Such ranking assists language users in debugging their models by providing direct links to the location of the faulty elements. Indeed by double-clicking on a row, the corresponding model element will be revealed in a new window.

For example, if we chose Phi as a concrete SBFL technique, it calculates the first rank for the second ModuleAssignment of the second if condition of the xArduino model of Figure 2.2 on page 12 where the defect is located (label 4). Therefore, the rank for the faulty element is correctly calculated to 1. However, there is also another element with the same rank. This is a common output returned

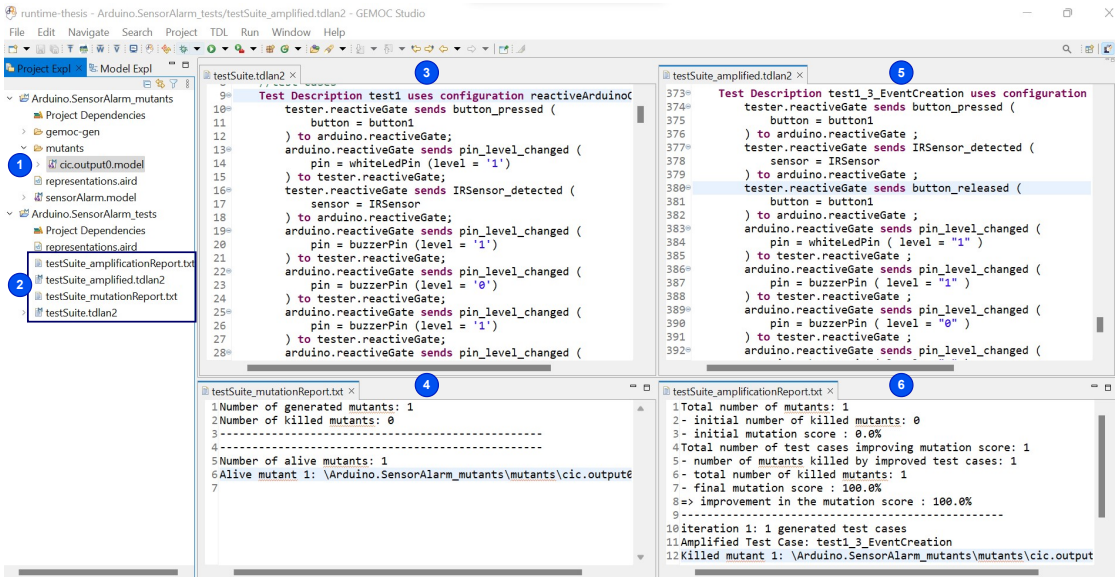


Figure 5.11: A screenshot of the results produced by our test amplification tool

by SBFL techniques, due to the so-named *tied* elements [150]. One common tie-breaking strategy for software programs is ordering based on line numbers in a text editor [150] and we support ordering based on model element position in a tree editor (i. e., provided by Ecore model editor).

5.6.2 Amplification Tool

We implemented our proposed amplification approach as part of the GEMOC studio [29], in Java. The source code is available on a public GitLab instance[4]. Using the provided tool, a domain expert can select an existing TDL test suite and run the amplification process on it. For example, we applied the tool on a TDL test suite for the xArduino sample model (label 1) that includes one test case (label 3). Figure 5.11 shows the results produced by the tool (label 2):

1. the mutation analysis result for the initial test suite, including the mutants killed by each of the test cases, and the list of alive mutants (label 4).
2. the amplified test suite comprising the initial test cases and the new test cases (label 5).
3. the mutation analysis result of the amplified test suite, including the initial mutation score, the final mutation score, the mutants killed by each new test case, and the mutants that are still alive after the amplification (label 6).

[4] https://gitlab.univ-nantes.fr/naomod/faezeh-public/xtdl/-/tree/master/test_amplification

Please note that files (2) and (3) will be generated if at least one new test case is produced by the test amplifier.

5.7 Evaluation

In this section, we first present our evaluation of the proposed debugging approaches (Section 5.7.1 on page 119). Next, we introduce an empirical evaluation designed and performed for the proposed test amplification approach (Section 5.7.2 on page 122).

5.7.1 Evaluation of Debugging Approaches

Our main objective in evaluating the debugging approaches is to validate their *genericity* regarding the supported xDSLs. Accordingly, our first research question is similar to RQ1 of our evaluation in Section 4.5 on page 93 (for evaluating the genericity of the proposed quality measurement techniques) which is:

RQ1: How much genericity do the debugging approaches provide regarding the supported xDSLs?

In addition, as the proposed *Fault Localization* component is offering an automatic debugging facility, we aim to investigate its validity by answering the following research question:

RQ2: How well can the proposed fault localization component find defects in a set of models?

In the following, we describe the experiment setup, answer the RQs, and describe threats to the validity of the experiments.

5.7.1.1 Experiment setup

Setup for RQ1. For RQ1, we aim to investigate whether the approaches can be used for different xDSLs. Accordingly, we chose the four different xDSLs used in Section 4.5.1 on page 94, including xFSM and xMiniJava non-reactive xDSLs, and xArduino and xPSSM reactive xDSLs. In particular, we used 45 test cases of five xFSM models, 77 test cases of six xMiniJava models, 22 test cases of six xArduino models, and 156 test cases of five xPSSM models, altogether 300 TDL test cases for 22 conforming models (as given in Table 4.2 on page 94). This should be noted that we excluded the 60 standard xPSSM models from the evaluation because the

		Non-Reactive xDSLs		Reactive xDSLs		Total
		xFSM	xMiniJava	xArduino	xPSSM	
Models &Tests	Number of tested models	5	6	6	5	22
	Total number of test cases	45	77	22	156	300
Mutation Analysis	Number of mutation operators	5	113	36	30	184
	Number of generated mutants	289	181	394	324	1,188
	Number of killed mutants	194	120	375	308	997
SBFL	Number of fault localized mutants	194	119	370	304	987

Table 5.2: Evaluation data for fault localization

provided test suite for each xPSSM model only contains a single test case, while SBFL needs multiple test cases for each model to be effectively applied.

For evaluating the debugging approaches on each considered model, there must be at least one defect in the model and at least one of its related test cases must be failing. For this, we used the evaluation result of our proposed *Mutation Analysis* component, already presented in Section 4.5.2 on page 95, also shown in Table 5.2 on page 120. More specifically, from 1,188 mutants generated out of 22 models through the application of 184 mutation operators, we used those killed by the related TDL test suites i. e., 194 xFSM mutants killed by 45 test cases, 120 xMiniJava mutants killed by 77 test cases, 375 xArduino mutants killed by 22 test cases, and 308 xPSSM mutants killed by 156 test cases, for a total of 997 killed mutants.

Setup for RQ2. The second research question targets the validity of the fault localization component. This needs to assess whether our fault localization component correctly ranks the faulty element of a model as first. Accordingly, we used the 1.088 killed mutants provided in the setup for RQ1, and to know the exact location of their injected fault, we used a tool named EMF Compare[5] to automatically find the faulty element of each mutant by comparing it with the original model. This should be noted that we are not aiming for an empirical evaluation of the performance of the different SBFL techniques and leave this to future work.

5.7.1.2 Evaluation result

Answering RQ1. As the *Interactive Debugging* component offers a manual debugging facility for the domain expert, we also evaluated it manually. We used it on both a set of failed TDL test cases of non-reactive models (such as the test cases for the xFSM and xMiniJava models) and a set of failed event-driven TDL test cases of reactive models (such as the test cases for the xArduino and xPSSM models). In both cases, we effectively debugged the model under test in unison with its test case, which ultimately helped us to localize the defect of the model.

[5] https://www.eclipse.org/emf/compare/

To perform fault localization using the SBFL facility, we run the 300 test cases on the 997 killed mutants, and then we successfully used the SBFL techniques to get the suspiciousness ranking of the elements of the mutants. As the examined models/mutants are defined using different xDSLs, it gives us the confidence to conclude that the approaches provide the expected genericity feature.

Answering RQ2. The second research question targets the validity of our proposed *Fault Localization* component. For answering this question, we need to investigate whether it can correctly find the faulty element of each mutant. Accordingly, we checked the result of running the SBFL techniques on 997 killed mutants from RQ1 to see the rank of the mutants' faulty element calculated by each SBFL technique. We observed that for 987 examined mutants (99 %), there is at least one SBFL technique that calculated the rank of its faulty element as first, hence showing the validity of our fault localization component.

As said earlier, we leave the performance evaluation of the different techniques as subject to future work, but we could show that in principle the proposed facility allows employing SBFL techniques for xDSLs. It is also worth mentioning that, since the fault localization component uses the test results and the coverage measurements produced by the TDL Interpreter and the coverage computation components (i. e., proposed in Sections 3.5 on page 62 and 4.2 on page 80, respectively), by answering RQ2, we are also validating the effectiveness of those two other components.

5.7.1.3 Threats to validity

One common external threat to the validity of our evaluation is that the debugging approaches might not work as expected for other modeling languages or in other language workbenches (also mentioned in Sections 3.8.4 and 4.5.3 on pages 77 and 97).

While answering the second research question in Section 5.7.1.2 on page 120, we observed that SBFL techniques can find the faulty element in a model. However, it is not clear which technique outperforms the other. This requires a deeper comparison between different SBFL techniques in an empirical evaluation to investigate their efficiency. This could also be useful in recommending the best techniques that offer the best ranking of the faulty elements given our coverage measurements.

In a recent survey on software fault localization [150], it is mentioned that SBFL is incapable of locating bugs that are caused by missing code. Accordingly, we ignored the mutation operators that define faults as the removal of models' elements and all the generated mutants have faults resulting from modification. This threatens the validity of our fault localization facility for models that are faulty because of missing elements. To overcome this threat, we should extend the facility with other fault localization techniques in the future.

Table 5.3: Setup for evaluating test amplification approach

		xArduino	xPSSM
Models & Tests	Number of tested models	6	65
	Size range of models (#EObjects)	18-59	13-154
	Initial test suite size (#test cases)	22	216
	#generated mutants	394	12,087

5.7.2 Evaluation of the Test Amplification Approach

Next, we evaluate our proposed test amplification approach, aiming to answer the following research questions (RQs):

RQ1 How much genericity is provided by the provided approach in terms of the supported reactive xDSLs?

RQ2 To what extent do the generated test cases increase the mutation score of the original, manually-written, test cases?

RQ3 To what extent do the *size* and the *quality* of the original test suites impact the amplification result?

In the following, we describe the experiment setup, answer the RQs and describe threats to validity of the experiments. The evaluation data is also accessible from a public GitLab repository[6].

5.7.2.1 Experiment setup

Setup for RQ1. For the first research question, we intend to evaluate the applicability of our approach to two different reactive xDSLs, xArduino and xPSSM, already presented in Section 3.8.1 on page 73. As our approach relies on mutation analysis, we used the sets of mutation operators defined in Section 4.5.1 on page 94 for each xDSL i. e., 36 for xArduino and 30 for xPSSM. For each xDSL, we used the set of models provided by Section 3.8.2 on page 75. As summarized in Table 5.3, 6 xArduino models and 65 xPSSM models (60 standard xPSSM models + 5 manually defined models).

Setup for RQ2. The second research question aims to assess the ability of the proposed approach for improving manually-written test cases. For this, we used the test cases provided by Section 3.8.2 on page 75. Cumulatively, 22 test cases for the xArduino models and 216 test cases for the xPSSM models, where 60 of

[6] https://gitlab.univ-nantes.fr/naomod/faezeh-public/xtdl/-/tree/master/publications-data/MODELS22-paper-data

them were the TDL versions of the test cases provided by the standard PSSM test suite [114].

As explained in Section 5.5.6 on page 114, our approach relies on mutation analysis to measure the degree of achieved improvement in the test suite quality. Accordingly, we used the WODEL mutant generator [60] to apply the defined mutation operators to the considered models. WODEL generated 394 and 12,087 mutants for the xArduino and the xPSSM models, respectively.

Setup for RQ3. The third research question aims to investigate whether the *size* and the *quality* of the initial test suite impacts the amplification result. In this regard, we classified our provided TDL test suites into four categories:

1. *Small Size Medium Quality (SSMQ)*: having one test case with mutation score $< 80\,\%$
2. *Small Size High Quality (SSHQ)*: having one test case with mutation score $\geq 80\,\%$
3. *Medium Size Medium Quality (MSMQ)*: having more than one test case with mutation score $< 80\,\%$
4. *Medium Size High Quality (MSHQ)*: having more than one test case with mutation score $\geq 80\,\%$

The rationale for selecting $80\,\%$ as the threshold to classify a test suite as having medium or high quality is because improving the mutation score beyond $80\,\%$ is significantly time-consuming for developers [129], and thus not necessarily worth the effort depending on the context.

For every category and xDSL, Table 5.4 presents the number of original test cases (column 3), the number of generated mutants (column 4), the number of mutants killed by the original test cases (column 5), and the original mutation score (column 6). The provided numbers are cumulative numbers, and the scores are the average scores considering all provided models and test suites. It should be noted that, when possible, we intentionally reduced the mutation score of an MSHQ test suite to have a new version of it as MSMQ or SSMQ (i. e., if reducing the mutation score had resulted in keeping only one test case in the new version of the test suite).

5.7.2.2 Evaluation result

Answering RQ1. The purpose of the first research question is to assess whether the approach can amplify test cases for various xDSLs. To answer this question, we used the prototype presented in Section 5.6.2 on page 118 for the two considered

Table 5.4: Evaluation result for amplifying test suites with different sizes and qualities

		# Orig. test cases	# Generated mutants	# Killed mutants by orig. test cases	Orig. mutation score	# New ampl. test cases	# New killed mutants by ampl. tests	Mutation score improvement	Regression test suite size (# orig. + # ampl.)	Regression test suite global score (orig. + improv.)
SSMQ	**xArduino**	4	148	60	44.08%	8	65	42.85%	12	86.93%
	xPSSM	43	5,653	4,022	71.99%	101	400	8.69%	144	80.68%
SSHQ	**xArduino**	1	15	13	86.67%	1	2	13.33%	2	100.00%
	xPSSM	18	2,205	1,873	84.38%	36	133	7.13%	54	91.51%
MSMQ	**xArduino**	2	231	106	45.89%	3	125	54.11%	5	100.00%
	xPSSM	44	4,229	3,385	74.23%	55	459	14.26%	99	88.49%
MSHQ	**xArduino**	6	241	222	91.10%	3	19	8.90%	9	100.00%
	xPSSM	156	4,453	4,249	93.28%	47	88	3.17%	203	96.45%

xDSLs and executed the test amplification tool on the 71 test suites. For 67 of them, new test cases were successfully generated—a total of 244—improving their original mutation score between 3.17 % and 54.11 % on average, based on the initial setup (detailed results are given shortly after, when answering the next research questions). Therefore, we can conclude that the approach does provide a certain level of genericity, i. e., the approach is not solely dedicated to a single specific xDSL, and can be applied to at least two different ones.

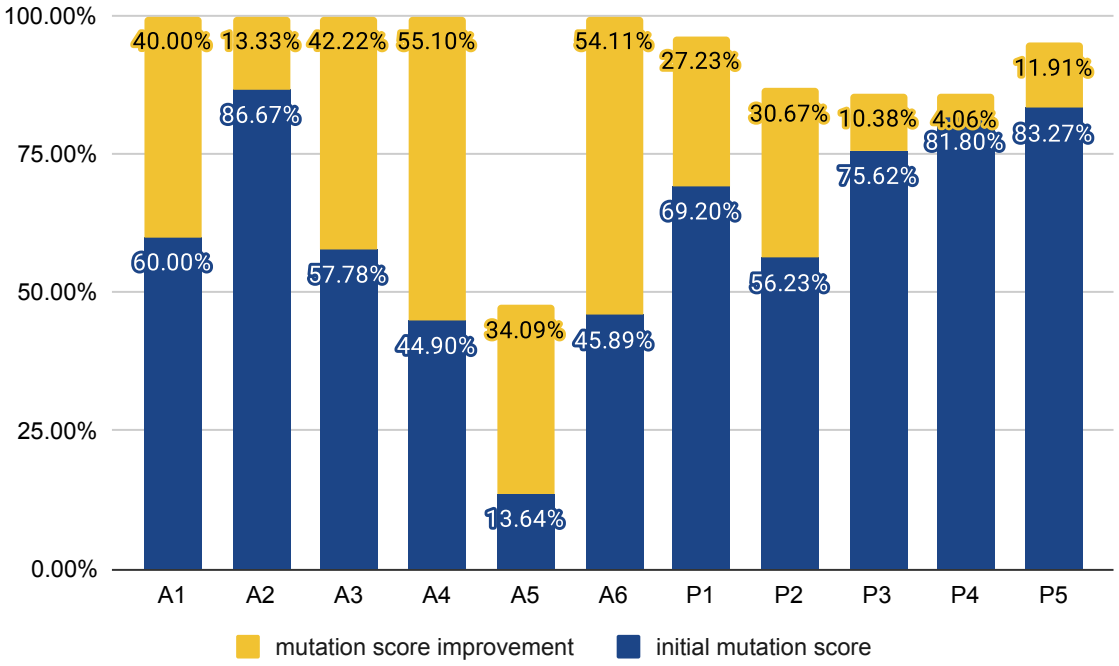
It is also worth mentioning that, to support an additional xDSL with our test amplification approach, the only additional cost for the language engineer is to define a set of mutation operators for the xDSL. These operators are defined once and can be reused for other purposes, such as test quality measurement [47, 67] and fault localization based on mutation analysis [150].

Answering RQ2. To answer the second research question, we must evaluate the degree of improvement provided by the generated test cases. Figure 5.12 presents the results for the selected 71 test suites using bar charts. Figure 5.12(a) displays

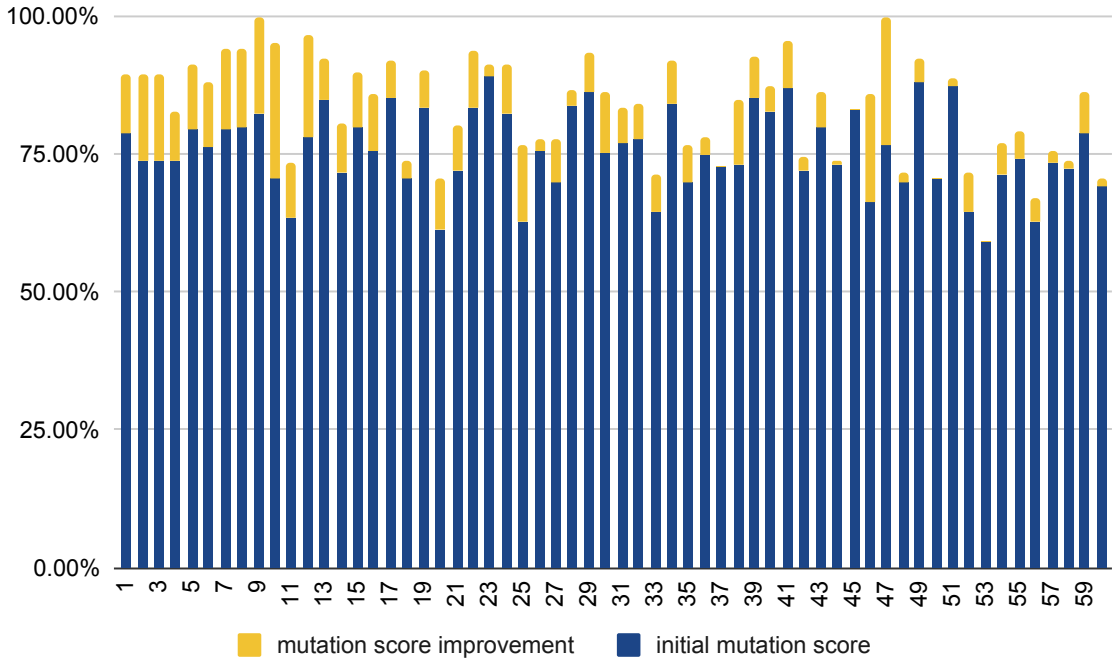
the result for the 11 test suites of manually defined models, that is, 6 xArduino models (A bars) and 5 xPSSM models (P bars). We have obtained mutation score improvements for all test suites, ranging from 4.06 % (P4) to 55.10 % (A4). Additionally, the final mutation score for 5 test suites (A1-A4 and A6) reaches 100 % after amplification. Figure 5.12(b) shows the result for the 60 test suites from the PSSM standard [114]; where each test suite has only one test case. Except for 4 cases (bars 37, 45, 50 and 53), the mutation score is improved, reaching 100 % for 2 test suites (bars 9 and 47). This also means that, even when starting from small test suites with just one test case, the approach is able to provide improvement. These results reveal the success of our test amplification approach in improving the mutation score of the models' test suites. However, the rate of improvement is different case by case, as we will discuss while answering RQ3.

Answering RQ3. Given a test suite to be amplified, its *size* (i. e., number of test cases) and *quality* (i. e., mutation score) may influence the level of improvement that our approach provides. The third research question targets this matter and to answer it, we run the experiment in four different setups. Table 5.4 lists detailed numbers related to our experiment. First, we compare the results for the test suites of the same size but with different qualities (SSMQ vs SSHQ, and MSMQ vs MSHQ). As can be seen, the number of new test cases (column 7) and the average improvement (column 9) for high-quality tests is less than the one of medium-quality tests. This is due to the fact that high-quality test suites need less improvement. However, the final mutation score of the regression test suite (last column) – the sum of the original score and the score improvement – is higher for high-quality tests. For example, for xArduino, the mutation score of SSMQ test suites is improved from 44.08 % to 86.93 %, but for the SSHQ ones, it is improved from 86.67 % to 100 %. Note that the scores refer to the average score of all considered test suites in each category. Also for xPSSM, the mutation score for the SSMQ test suites improves from 71.99 % to 80.68 %, and for the SSHQ ones from 84.38 % to 91.51 %. This could imply that, when the original test cases have higher quality, there is more room for test amplification. By amplifying high-quality tests, it is more probable to generate new *effective* test cases.

Second, we compare the results for the test suites with different sizes but similar qualities (SSMQ vs MSMQ, and SSHQ vs MSHQ). According to the numbers in the last column, the final mutation score is higher when the original test suite has more test cases. For instance, comparing the final mutation score of the xArduino test suites, for SSMQ is 86.93 % while for MSMQ is 100 %, and for both SSHQ and MSHQ is 100 %. Likewise, for the xPSSM test suites, the final mutation score is 80.68 % for SSMQ but 88.49 % for MSMQ, and 91.51 % for SSHQ but 96.45 % for MSHQ. Therefore, for test suites with more test cases, it appears that there is



(a) Mutation score improvement for 11 test suites of manually defined models: 6 xArduino models (A bars) and 5 xPSSM models (P bars)



(b) Mutation score improvement for 60 existing test suites from the PSSM standard [114]

Figure 5.12: Mutation score improvement by test amplification

more chance to generate new test cases improving the mutation score. A possible explanation is that our approach runs the amplification on *every* test case of the original test suite, each time by applying all the possible modifiers to generate as many new test cases as possible. Hence, the more available initial test cases, the more generated test cases, and the more chances for improvement of the test suite quality.

5.7.2.3 Threats to validity

In addition to the external threat to the validity of our evaluation regarding the considered xDSLs (i. e., mentioned several times in Sections 3.8.4, 4.5.3, and 5.7.1.3 on pages 77, 97, and 121), we list here multiple other threats. First, we kept iterating the amplification process while improvements were observed, meaning that the input for the next iteration is the output of the current iteration. Meaning that among all the new test cases generated in the current iteration, we only keep those improving the mutation score for the next iteration (label 4 in Figure 5.4 on page 109). Without this test case selection criterion (i. e., iterating from label 3 of Figure 5.4 on page 109), the number of generated test cases increases exponentially after a few iterations. However, the non-effective test cases (i. e., test cases not contributing to improving the mutation score in the current iteration) might become effective in the next iterations since a combination of several modifiers is applied to them. It is also worth mentioning that we experimented the tool for up to 3 iterations while the mutation score is less than 100 % to avoid a huge experimentation time; nonetheless, we may reach higher mutation scores with more iterations. Note that the users of the tool are allowed to change this stop criterion.

Next, we used the modifiers presented in Section 5.5.3 on page 110. Other more complex modifiers could be devised, which may show more effectiveness in fault localization. This should be investigated in the future, but we have shown that our proposed modifier set is enough to produce effective amplified test cases.

Moreover, as discussed in Section 5.5.2 on page 109, we assume that a set of mutation operators has been provided for each considered xDSL. Depending on the quality of these operators (e. g., their metamodel footprint) the diversity of the generated mutants would differ, leading to variations in the effectiveness of the test amplification tool. It would be interesting to consider other test case selection criteria such as increasing coverage.

Lastly, in the best scenario, amplified test cases must be approved by the developers who wrote the original test cases [42]. Accordingly, there is a need for a user-centric evaluation to assess the value of the generated TDL test cases from the user's perspective.

5.8 Conclusion

In this chapter, we have presented facilities for debugging and improving test cases of executable models. We offered approaches for both manual and automatic debugging of models' test cases as well as a test amplification approach targeting the regression testing of models. The proposed facilities are generic and applicable to any xDSL and are evaluated on several different xDSLs. The evaluation result showed the benefits of the facilities in debugging failed test cases and improving them in finding potential faults (as given by the mutation score).

Discussions and Improvements. In this chapter, we demonstrated the effectiveness of the SBFL techniques for localizing faults of the executable models. However, a deeper investigation is required to identify the most efficient ones in the context of xDSLs. Moreover, the proposed test amplification approach can be expanded with more advanced features. For example, input modifiers changing event sequences may need to create dynamic objects following certain criteria which can be offered using search-based techniques, such as MOMoT [26]. It is also beneficial to make explicit and extensible the set of input modifiers, possibly via a dedicated DSL.

In general, the result of the test amplification approach can be influenced by several different parameters such as the kind of input modifiers, their application policy, the number of iterations of the amplification process, the test oracle generation technique, and the test selection criteria and more empirical evaluation are needed to measure the impact of each.

Chapter 6

Conclusion and Perspectives

6.1 Conclusion

An executable DSL with testing support allows its users, i. e., domain experts, to ensure the correctness of their modeled behavior as early as possible. However, providing a testing framework for an xDSL faces several challenges. First, to enable the domain experts to accomplish testing of their models, a testing language is required that supports the use of domain concepts in the definition of test cases, the execution of test cases on the models, and the comparison of models' behavior against their test case expectations. As the domain concepts, the model execution facilities, and the observable behavior of executed models differ from one xDSL to another, providing such a testing language for each and every xDSL is a costly and challenging endeavor. Moreover, to perform model testing *efficiently*, a testing framework should provide at least three other important features: test quality measurement to evaluate the strength of test cases, test debugging to find the cause of test failure, and test improvement to strengthen the written test cases when needed.

In this thesis, we proposed a generic testing framework for xDSLs that systematically realizes each of the aforementioned needs for any given xDSL. At its core, the framework relies on the standardized Test Description Language (TDL), and can automatically generate a TDL library for a considered xDSL. This library allows domain experts to define executable test cases for conforming models. We also provided a test execution engine dedicated to running TDL test cases on executable models which also generates the test execution results. In addition to these essential facilities, we proposed three supplementary services to help the domain experts in *efficiently* testing models. First, we provided generic test quality measurement facilities for xDSLs. In particular, we proposed a generic approach for computing the *coverage* of TDL test cases and we integrated our approach

with a model mutation framework [62] to offer *mutation analysis* to any xDSL. Second, for debugging failed test cases, we introduced both a manual debugging approach based on interactive debugging and an automatic fault localizer using SBFL techniques. Third, to automatically improve the quality of those written test cases that are not strong in finding regression faults, we proposed a generic test amplification approach. From an existing test suite, this approch produces new test cases that improve input test suite in detecting regression faults.

To sum up, this thesis supports language engineers in providing comprehensive testing facilities for their xDSLs in a systematic manner. Once the proposed framework is applied to a given xDSL, the xDSL users (i. e., domain experts) can define test cases for the conforming models, run test cases on models and get the results, measure the quality of test cases, debug the failed test cases, and improve the written test cases for regression testing.

6.2 Limitations and Possible Improvements

In this section, we introduce the limitations we identified in the work presented in this thesis, along with possible solutions for improving them.

Genericity regarding supported xDSLs. The genericity of the framework is assessed by its application on six different DSLs, but to assure its practicality for other modeling languages, more experiments are needed. For example, evaluating the framework on xDSLs with a larger amount of dynamic features and consequently, for testing models with more complex runtime states. In addition, while at the moment we tried testing executable models each conforming to a single xDSL, it would be very interesting to study the possible use of the framework on xDSLs that are a composition of several interconnected xDSLs, which means the test execution engine must be able to interact with several connected models that are co-executing. Lastly, we defined our framework considering the GEMOC Studio as a reference for the xDSL implementation and we built our tool on top of it. In fact, the GEMOC Studio only allows building DSLs based on Ecore, and using a handful of metaprogramming approaches such as Kermeta [78], Xtend [49], and ALE [92], among others. In the literature, there are also other language workbenches [51], and investigating the portability of the provided framework on other workbenches would be beneficial.

Usability. In this thesis, two roles were considered, the language engineers who use the proposed framework to offer testing facilities for their xDSLs and the domain experts who are the end users of the provided facilities. We aimed at moderating the endeavor of language engineers by proposing a generic testing framework as

well as easing the model testing activities for domain experts by supporting domain concepts and providing automation. However, a user study must be conducted to evaluate the framework usability from the point of view of both considered roles. This user study is also essential for the parts of our contribution related to test amplification because as discussed by Danglot et al. [42], amplified test cases must be approved by the developers of the original test cases who are the domain experts in the context of this thesis.

Reliability of mutation analysis result. In this thesis, we used mutation analysis for both measuring the ability of test cases in detecting potential faults (Section 4.3 on page 89) and selecting the most efficient amplified test cases regarding fault detection improvement (Section 5.5 on page 106). Therefore, it is important to obtain reliable mutation analysis results. However, this result may vary subject to the mutation operators because depending on the quality of the operators (e. g., their metamodel footprint), the diversity of the generated mutants would differ, leading to variations in the mutation analysis result. To deal with such a situation, one possible solution could be a systematic generation of mutation operators for any given xDSL. For example, Troya et al. [140] investigated systematic mutation of ATL model transformations and this research can be followed but for the context of xDSLs. As mutation operators perform small modifications to a model to generate mutants, an interesting solution could be inferring them utilizing model editing operations for metamodels as introduced by Kehrer et al. [83].

Efficiency of SBFL techniques. A common output returned by SBFL techniques is detecting several elements with the same suspiciousness, called to be *tied* to the same position in a ranking [150]. As model elements ranking is used by the domain experts to find faulty elements, the more tied elements lead to more effort for detecting faults. The *best* case scenario is when the first tied element is faulty and the *worst* case scenario is when the last tied element has the fault since all the other tied elements must be examined to find the fault. One common tie-breaking strategy for software programs is ordering based on line numbers in a text editor [150] and we support ordering based on model element position in a tree editor (i. e., provided by Ecore model editor). However, there are other techniques such as confidence-based and data dependency-based strategies which can be considered in the future to improve efficiency.

Another concern lies in the choice of the SBFL technique. As already mentioned in Section 5.6.1 on page 116, there are approximately 30 SBFL techniques in the literature [110, 131, 150] and we currently implemented 18 of them in our framework. Currently, the domain expert must manually choose a random technique from the list of available techniques and it is not clear which technique can produce a better

ranking of elements. To efficiently apply SBFL techniques in the context of xDSLs and to recommend the best technique to the domain expert for a given model, we need to know which technique outperforms the other for a given xDSL or a given model. Grasping such information requires a deeper comparison between different SBFL techniques in an empirical evaluation. An investigation is needed to find out if there is a meaningful relationship between an SBFL technique and the features of a given xDSL such as its application domain, the structure of its abstract syntax, its semantics and the kind of behavior that can be expressed by its conforming models, and the kind of defects that may occur in its conforming models.

The impact of different parameters in amplification results. In our proposed test amplification approach (Section 5.5 on page 106), there are several parameters that may affect the final result. First, we have defined a set of test data modifiers to generate new test input data and we obtained promising results in our evaluation. However, an empirical evaluation is needed to measure the effectiveness of each modifier both individually and in combination with other modifiers. This can be achieved by analyzing the final amplified test cases and investigating the kind of modifiers contributing to generating them and the kind of mutants killed by each amplified test case. Second, we experimented with the tool for up to 3 iterations while the mutation score is less than 100 % to avoid a huge experimentation time; nonetheless, we may reach higher mutation scores with more iterations. Third, we do not currently measure the execution overhead of the amplification tool and it is beneficial to investigate in which situations the provided improvement is worth the execution overhead.

6.3 Perspectives

In addition to the possible improvements for resolving the existing limitations, we can vision several perspectives to continue the work of this thesis in the future.

Testing support for compiled executable DSLs. As already mentioned in Section 2.2.3 on page 13, the execution semantics of DSLs can be defined as translational (i. e., comilation) or operational (i. e., interpretation). This thesis focused on DSLs with operational semantics as our objective was to offer early testing of models using the domain concepts. Supporting compiled DSLs can widen the scope of our contributions to many other DSLs, however, the challenge is the dependency of these DSLs on an arbitrarily different target language. In particular, as all their execution facilities are indeed relying on those of their target language, the dynamic V&V techniques can be performed on the target language, thus using the target domain concepts. This challenge is also discussed by Bousse et al. for

domain-level observation and control of compiled DSLs [33]. They propose the definition of a feedback manager for the compiled DSLs that can translate the execution steps and states of a target language back into the source domain. It is apparent that on the basis of their approach, it is possible to offer testing facilities for compiled DSLs as well, but a challenge remains unsolved: the translation of test input data (e. g., input events) towards the target domain, and likewise for output data of the target domain back to the source domain.

Broadening test oracle definition approaches. Our provided facilities for test case definition and execution allow the specification of test oracles as to what should be accepted by a model and how the model should react. Also, they are de facto constrained by how test cases are made in TDL i. e., as a sequence of expected output (e. g., expected events). However, there are several interesting other ways of defining test oracles for asserting different kinds of behavior, such as specifying what should *not* be accepted/retrieved by/from a model. Also for testing models with concurrent execution, test oracles are commonly defined as temporal properties (e. g., using Linear Temporal Logic (LTL) [23] or OCL queries extended with temporal logic [158]) and runtime monitoring is needed to evaluate them. In a recent work by Leroy et al. runtime monitoring is offered generically for xDSLs considered in the scope of this thesis [94], hence arising a new opportunity for extending the proposed test case definition and execution facilities.

New coverage metrics, rules, and facilities. There are many coverage metrics in the literature for software programs [12] and some are adapted for particular DSLs, such as logic coverage for State Machines [50], data-flow coverage for executable UML models [145], branch coverage for activity diagrams [27], among many others. On the other hand, to overcome the xDSLs heterogeneity challenge, we introduced a new generic coverage metric based on model elements which is applicable to any model. However, this metric may not offer the same benefits as some existing coverage metrics such as branch coverage, which gives information on traversed execution paths. An interesting line of research would therefore be to provide a systematic approach to adapt different existing coverage metrics for any given xDSL.

Another noteworthy point regarding coverage is the definition of DSL-specific coverage rules. The coverage computation service of the framework supports language engineers in defining them by proposing a dedicated metalanguage. This metalanguage allows the definition of both inclusion and exclusion coverage rules, but the technique used for specifying the inclusion and exclusion conditions is now limited to containment relationships between elements. One language engineer may need to specify a specific condition based on several parameters e. g., querying

the runtime state of an executed model to specify a particular state upon which a coverage rule must be applied. This requires an extension of the proposed metalanguage to support the definition of more sophisticated coverage rules for DSLs.

Lastly, this thesis offered coverage computation for both measuring the quality of test cases and applying SBFL fault localization techniques. However, many other coverage-based techniques can be envisioned for the future, such as test minimization, test selection [10], and test prioritization [108]. Additionally, coverage improvement can also be considered as test selection criteria of our test amplification approach (instead of mutation analysis) as discussed in the following.

Test amplification for other objectives. According to a recent survey on test amplification [42], amplifying test cases is performed hitherto for different objectives, such as improving coverage, reproducing crashes, detecting new faults, and localizing existing faults, among others. Accordingly, various techniques are applied so far, including: analyzing the body of test cases and/or the system under test, modifying the test execution process, concolic and symbolic execution, and search-based heuristics [42]. In this thesis, we aimed at improving the quality of test cases in detecting new potential faults by generating new test cases through analyzing test cases and their model under test. An interesting research line would be extending the proposed test amplification approach for other objectives such as improving coverage. Also, the approach can be recast as a search process based on genetic algorithms. This way, crossover operators for the TDL test cases would need to be defined, input modifiers would be used as mutation operators, and the fitness function would be driven by mutation score improvement. It can be then investigated which strategy has a better result and/or performance.

Automatic test case generation. The main difference between test case generation techniques and test amplification approaches is the kind of input they use for the generation of test cases. The former uses the specification of the system under test while the latter functions on the existing manually-written test cases. As a model is indeed a specification of a system, a plenty of model-based test case generation techniques are already introduced in the literature [108]. To advance our proposed framework with automatic test case generation, we need to discover the state-of-the-art for generic techniques that can be applied to different models.

Automatic co-evolution of models and test cases. In the realm of programming languages, a software program and its test cases are usually both implemented using the same language (such as Java programs and their JUnit test cases). Accordingly, when the program evolves, the compiler can detect inconsistencies in its

related test cases. In the context of this thesis, the test case and the model under test are two different executable models conforming to two different languages and this raises co-evolution challenges at two different levels: language level and model level. Considering possible evolutions of a DSL implementation (i) changes in its abstract syntax should be propagated to the TDL data types generated from it; (ii) changes in its runtime state definition might require updates of the test oracles defined based on the runtime state; (iii) for reactive xDSLs, changes in its behavioral interface might need updates of the test oracles defined based on it; and (iv) changes in its execution rules may impact the test case execution result as the model execution trace could be affected, consequently, the results of all other parts (i. e., quality measurement, debugging, and amplification) must be verified again. The only available solution is re-executing the framework after each change, in particular running the *TDL Library Generator* for updating the TDL data types and running the other facilities for generating new results. However, changes in the oracles must be recognized and performed manually by the domain experts at the moment. They are also in charge of co-evolving models and their test cases, meaning that when there is a change in a model, its related test cases need to be checked manually to assure its compatibility with the recent changes in the model. It is apparent that as the size and the number of models and their test cases increase, co-evolving them becomes more difficult, and automated solutions turn necessary [84]

Appendix A

Ecore to TDL Transform Rules

```
1  -- @nsURI Ecore=http://www.eclipse.org/emf/2002/Ecore
2  -- @path TDL=/org.etsi.mts.tdl.model/model/tdl.ecore
3
4  module ecore2tdl;
5  create OUT : TDL from IN : Ecore;
6
7  helper def: tokenNames: Sequence(String) = Sequence {
8      'Package', '{', '}', 'with', 'perform', 'action', '(', ',', ')', 'on',
       'test', 'objectives', ':', ';', 'name', 'time', 'label', 'constraints
      ', 'Action', 'alternatively', 'or', 'Annotation', '*', '?', '=', '
      assert', 'otherwise', 'set', 'verdict', 'to', '→', '[', ']', 'times',
       'repeat', 'break', 'Note', 'create', 'of', 'type', 'bind', 'Component
      ', 'Type', 'having', 'if', 'else', 'connect', 'as', 'Map', 'in', '.',
      'new', 'containing', 'Use', 'Signature', 'Collection', 'default', '+',
       '-', '/', 'mod', '>', '<', '≥', '≤', '==', '!=', 'and', 'xor', 'not'
      , 'size', 'Import', 'all', 'from', 'Function', 'returns', 'instance',
      'returned', 'Predefined', 'gate', 'Gate', 'accepts', 'sends', '
      triggers', 'calls', 'responds', 'response', 'interrupt', 'optional', '
      mapped', 'omit', 'argument', 'optionally', 'run', 'parallel', '
      parameter', 'every', 'component', 'is', 'quiet', 'for', 'terminate', '
      where', 'it', 'assigned', 'Test', 'Configuration', 'Description', '
      Implementation', 'uses', 'configuration', 'execute', 'bindings', '
      Objective', 'description', 'Time', 'out', 'timer', 'start', 'stop', '
      variable', 'waits', 'extends', 'SUT', 'Tester', 'Message', 'Procedure'
      , 'In', 'Out', 'Exception', 'last', 'previous', 'first'
9  };
10 helper def: enums: Set (Ecore!EEnum)= Ecore!EEnum→allInstances()→asSet()
       ;
11 helper def: enumLiterals: Set (Ecore!EEnumLiteral) = thisModule.enums→
      collect(e | e.eLiterals);
12 helper context Ecore!EPackage def: basicDataTypes: Set(Ecore!EDataType) =
       Ecore!EDataType→allInstances()→
13     select(type | not type.oclIsKindOf(Ecore!EEnum))→asSet();
```

```
14
15 helper def: dynamicAnnotationType: TDL!AnnotationType = OclUndefined;
16 helper context Ecore!EStructuralFeature def: isDynamicFeature: Boolean =
17     if (self.eAnnotations→select(a | a.source = 'dynamic' or a.source = '
    aspect')→notEmpty()) then true else false endif;
18 helper context Ecore!EClass def: isDynamicClass: Boolean =
19     if (self.eAnnotations→select(a | a.source = 'dynamic' or a.source = '
    aspect')→notEmpty()) then true else false endif;
20 helper def: abstractAnnotationType: TDL!AnnotationType = OclUndefined;
21
22 rule simpleConcreteStaticClass2Type{
23     from class: Ecore!EClass (class.eAllStructuralFeatures.isEmpty() and
    not class.abstract and not class.isDynamicClass)
24     to type: TDL!StructuredDataType(
25         name ← if (thisModule.tokenNames.includes(class.name))
26                 then '_' + class.name
27                 else class.name
28                 endif)
29     do{
30         thisModule.tokenNames ← thisModule.tokenNames.append(type.name.
    toString());
31     }
32 }
33 rule simpleConcreteDynamicClass2Type{
34     from class: Ecore!EClass (class.eAllStructuralFeatures.isEmpty() and
    not class.abstract and class.isDynamicClass)
35     to type: TDL!StructuredDataType(
36         name ← if (thisModule.tokenNames.includes(class.name))
37                 then '_' + class.name
38                 else class.name
39                 endif,
40         annotation ← dynamicAnnotation),
41         dynamicAnnotation: TDL!Annotation(
42             key ← if (thisModule.dynamicAnnotationType.oclIsUndefined())
43                     then thisModule.eAnnotation2annotationType()
44                     else thisModule.dynamicAnnotationType
45                     endif,
46             annotatedElement ← type)
47     do{
48         thisModule.tokenNames ← thisModule.tokenNames.append(type.name.
    toString());
49     }
50 }
51 rule simpleAbstractStaticClass2Type{
52     from class: Ecore!EClass (class.eAllStructuralFeatures.isEmpty() and
    class.abstract and not class.isDynamicClass)
53     to type: TDL!StructuredDataType(
54         name ← if (thisModule.tokenNames.includes(class.name))
55                 then '_' + class.name
```

```
56                  else class.name
57                  endif,
58          annotation ← abstractAnnotation),
59          abstractAnnotation: TDL!Annotation(
60              key ← if (thisModule.abstractAnnotationType.oclIsUndefined())

61                      then thisModule.abstract2annotationType()
62                      else thisModule.abstractAnnotationType
63                      endif,
64              annotatedElement ← type)
65      do{
66          thisModule.tokenNames ← thisModule.tokenNames.append(type.name.
    toString());
67      }
68  }
69  rule simpleAbstractDynamicClass2Type{
70      from class: Ecore!EClass (class.eAllStructuralFeatures.isEmpty() and
    class.abstract and class.isDynamicClass)
71      to type: TDL!StructuredDataType(
72          name ← if (thisModule.tokenNames.includes(class.name))
73                  then '_' + class.name
74                  else class.name
75                  endif),
76          abstractAnnotation: TDL!Annotation(
77              key ← if (thisModule.abstractAnnotationType.oclIsUndefined())

78                      then thisModule.abstract2annotationType()
79                      else thisModule.abstractAnnotationType
80                      endif,
81              annotatedElement ← type),
82          dynamicAnnotation: TDL!Annotation(
83              key ← if (thisModule.dynamicAnnotationType.oclIsUndefined())
84                      then thisModule.eAnnotation2annotationType()
85                      else thisModule.dynamicAnnotationType
86                      endif,
87              annotatedElement ← type)
88      do{
89          thisModule.tokenNames ← thisModule.tokenNames.append(type.name.
    toString());
90      }
91  }
92  rule featuredConcreteStaticClass2Type{
93      from class: Ecore!EClass (class.eAllStructuralFeatures.notEmpty()
94                                  and class.eSuperTypes.isEmpty()
95                                  and not class.abstract
96                                  and not class.isDynamicClass)
97      to type: TDL!StructuredDataType(
98          name ← if (thisModule.tokenNames.includes(class.name))
99                  then '_' + class.name
```

```
100                 else class.name
101                 endif,
102         member ← class.eStructuralFeatures→ collect(feature |
103                 if (feature.eAnnotations.notEmpty() and feature.
    isDynamicFeature)
104                     then thisModule.dynamicFeature2annotatedMember(
    feature)
105                     else thisModule.staticFeature2member(feature)
106                     endif))
107     do{
108         thisModule.tokenNames ← thisModule.tokenNames.append(type.name.
    toString());
109     }
110 }
111 rule featuredConcreteDynamicClass2Type{
112     from class: Ecore!EClass (class.eAllStructuralFeatures.notEmpty()
113                               and class.eSuperTypes.isEmpty()
114                               and not class.abstract
115                               and class.isDynamicClass)
116     to type: TDL!StructuredDataType(
117         name ← if (thisModule.tokenNames.includes(class.name))
118                 then '_' + class.name
119                 else class.name
120                 endif,
121         member ← class.eStructuralFeatures→ collect(feature |
122                 if (feature.eAnnotations.notEmpty() and feature.
    isDynamicFeature)
123                     then thisModule.dynamicFeature2annotatedMember(
    feature)
124                     else thisModule.staticFeature2member(feature)
125                     endif),
126         annotation ← dynamicAnnotation),
127         dynamicAnnotation: TDL!Annotation(
128             key ← if (thisModule.dynamicAnnotationType.oclIsUndefined())
129                     then thisModule.eAnnotation2annotationType()
130                     else thisModule.dynamicAnnotationType
131                     endif,
132             annotatedElement ← type)
133     do{
134         thisModule.tokenNames ← thisModule.tokenNames.append(type.name.
    toString());
135     }
136 }
137 rule featuredAbstractStaticClass2Type{
138     from class: Ecore!EClass (class.eAllStructuralFeatures.notEmpty()
139                               and class.eSuperTypes.isEmpty()
140                               and class.abstract
141                               and not class.isDynamicClass)
142     to type: TDL!StructuredDataType(
```

```
143         name ← if (thisModule.tokenNames.includes(class.name))
144                 then '_' + class.name
145                 else class.name
146                 endif,
147         member ← class.eStructuralFeatures→ collect(feature |
148                 if (feature.eAnnotations.notEmpty() and feature.
    isDynamicFeature)
149                     then thisModule.dynamicFeature2annotatedMember(
    feature)
150                     else thisModule.staticFeature2member(feature)
151                     endif),
152         annotation ← abstractAnnotation),
153         abstractAnnotation: TDL!Annotation(
154             key ← if (thisModule.abstractAnnotationType.oclIsUndefined())

155                     then thisModule.abstract2annotationType()
156                     else thisModule.abstractAnnotationType
157                     endif,
158             annotatedElement ← type)
159     do{
160         thisModule.tokenNames ← thisModule.tokenNames.append(type.name.
    toString());
161     }
162 }
163
164 rule featuredAbstractDynamicClass2Type{
165     from class: Ecore!EClass (class.eAllStructuralFeatures.notEmpty()
166                               and class.eSuperTypes.isEmpty()
167                               and class.abstract
168                               and class.isDynamicClass)
169     to type: TDL!StructuredDataType(
170         name ← if (thisModule.tokenNames.includes(class.name))
171                 then '_' + class.name
172                 else class.name
173                 endif,
174         member ← class.eStructuralFeatures→ collect(feature |
175                 if (feature.eAnnotations.notEmpty() and feature.
    isDynamicFeature)
176                     then thisModule.dynamicFeature2annotatedMember(
    feature)
177                     else thisModule.staticFeature2member(feature)
178                     endif)),
179         abstractAnnotation: TDL!Annotation(
180             key ← if (thisModule.abstractAnnotationType.oclIsUndefined())

181                     then thisModule.abstract2annotationType()
182                     else thisModule.abstractAnnotationType
183                     endif,
184             annotatedElement ← type),
```

```
185         dynamicAnnotation: TDL!Annotation(
186             key ← if (thisModule.dynamicAnnotationType.oclIsUndefined())
187                     then thisModule.eAnnotation2annotationType()
188                     else thisModule.dynamicAnnotationType
189                     endif,
190             annotatedElement ← type)
191     do{
192         thisModule.tokenNames ← thisModule.tokenNames.append(type.name.
    toString());
193     }
194 }
195
196 rule concreteInheritedStaticClass2Type{
197     from class: Ecore!EClass (class.eAllStructuralFeatures.notEmpty()
198                               and class.eSuperTypes.notEmpty()
199                               and not class.abstract
200                               and not class.isDynamicClass)
201     to type: TDL!StructuredDataType(
202         name ← if (thisModule.tokenNames.includes(class.name))
203                 then '_' + class.name
204                 else class.name
205                 endif,
206         member ← class.eStructuralFeatures→ collect(feature |
207                 if (feature.eAnnotations.notEmpty() and feature.
    isDynamicFeature)
208                     then thisModule.dynamicFeature2annotatedMember(
    feature)
209                     else thisModule.staticFeature2member(feature)
210                     endif),
211         extension ← class.eSuperTypes → collect (st | thisModule.
    superClass2extension(st)))
212     do{
213         thisModule.tokenNames ← thisModule.tokenNames.append(type.name.
    toString());
214     }
215 }
216
217 rule concreteInheritedDynamicClass2Type{
218     from class: Ecore!EClass (class.eAllStructuralFeatures.notEmpty()
219                               and class.eSuperTypes.notEmpty()
220                               and not class.abstract
221                               and class.isDynamicClass)
222     to type: TDL!StructuredDataType(
223         name ← if (thisModule.tokenNames.includes(class.name))
224                 then '_' + class.name
225                 else class.name
226                 endif,
227         member ← class.eStructuralFeatures→ collect(feature |
```

```
228                 if (feature.eAnnotations.notEmpty() and feature.
    isDynamicFeature)
229                     then thisModule.dynamicFeature2annotatedMember(
    feature)
230                     else thisModule.staticFeature2member(feature)
231                     endif),
232         extension <- class.eSuperTypes -> collect (st | thisModule.
    superClass2extension(st)),
233         annotation <- dynamicAnnotation),
234         dynamicAnnotation: TDL!Annotation(
235             key <- if (thisModule.dynamicAnnotationType.oclIsUndefined())
236                     then thisModule.eAnnotation2annotationType()
237                     else thisModule.dynamicAnnotationType
238                     endif,
239             annotatedElement <- type)
240     do{
241         thisModule.tokenNames <- thisModule.tokenNames.append(type.name.
    toString());
242     }
243 }
244
245 rule abstractInheritedStaticClass2Type{
246     from class: Ecore!EClass (class.eAllStructuralFeatures.notEmpty()
247                               and class.eSuperTypes.notEmpty()
248                               and class.abstract
249                               and not class.isDynamicClass)
250     to type: TDL!StructuredDataType(
251         name <- if (thisModule.tokenNames.includes(class.name))
252                 then '_' + class.name
253                 else class.name
254                 endif,
255         member <- class.eStructuralFeatures-> collect(feature |
256                 if (feature.eAnnotations.notEmpty() and feature.
    isDynamicFeature)
257                     then thisModule.dynamicFeature2annotatedMember(
    feature)
258                     else thisModule.staticFeature2member(feature)
259                     endif),
260         extension <- class.eSuperTypes -> collect (st | thisModule.
    superClass2extension(st)),
261         annotation <- abstractAnnotation),
262         abstractAnnotation: TDL!Annotation(
263             key <- if (thisModule.abstractAnnotationType.oclIsUndefined())

264                     then thisModule.abstract2annotationType()
265                     else thisModule.abstractAnnotationType
266                     endif,
267             annotatedElement <- type)
268     do{
```

```
269         thisModule.tokenNames ← thisModule.tokenNames.append(type.name.
    toString());
270     }
271 }
272 
273 rule abstractInheritedDynamicClass2Type{
274     from class: Ecore!EClass (class.eAllStructuralFeatures.notEmpty()
275                               and class.eSuperTypes.notEmpty()
276                               and class.abstract
277                               and class.isDynamicClass)
278     to type: TDL!StructuredDataType(
279         name ← if (thisModule.tokenNames.includes(class.name))
280                 then '_' + class.name
281                 else class.name
282                 endif,
283         member ← class.eStructuralFeatures→ collect(feature |
284                 if (feature.eAnnotations.notEmpty() and feature.
    isDynamicFeature)
285                     then thisModule.dynamicFeature2annotatedMember(
    feature)
286                     else thisModule.staticFeature2member(feature)
287                     endif),
288         extension ← class.eSuperTypes → collect (st | thisModule.
    superClass2extension(st))),
289         abstractAnnotation: TDL!Annotation(
290             key ← if (thisModule.abstractAnnotationType.oclIsUndefined())

291                     then thisModule.abstract2annotationType()
292                     else thisModule.abstractAnnotationType
293                     endif,
294             annotatedElement ← type),
295         dynamicAnnotation: TDL!Annotation(
296             key ← if (thisModule.dynamicAnnotationType.oclIsUndefined())
297                     then thisModule.eAnnotation2annotationType()
298                     else thisModule.dynamicAnnotationType
299                     endif,
300             annotatedElement ← type)
301     do{
302         thisModule.tokenNames ← thisModule.tokenNames.append(type.name.
    toString());
303     }
304 }
305 
306 lazy rule superClass2extension{
307     from class: Ecore!EClass
308     to parent : TDL!Extension(
309         extending ← class
310     )
311 }
```

```
312
313 lazy rule staticFeature2member{
314     from feature: Ecore!EStructuralFeature (feature.eAnnotations.isEmpty()
     or
315             (feature.eAnnotations.notEmpty() and (not feature.
    isDynamicFeature)))
316     to member: TDL!Member(
317         name ← if (thisModule.tokenNames.includes(feature.name))
318                 then '_' + feature.name
319                 else feature.name
320                 endif,
321         dataType ← if (feature.oclIsKindOf(Ecore!EAttribute) and
    thisModule.enums→excludes(feature.eType))
322                     then  thisModule.dataType2simpleType(feature.eType)
323                     else feature.eType
324                     endif)
325 }
326 lazy rule dynamicFeature2annotatedMember{
327     from feature: Ecore!EStructuralFeature (feature.eAnnotations.notEmpty
    () and feature.isDynamicFeature)
328     to member: TDL!Member(
329         name ← if (thisModule.tokenNames.includes(feature.name))
330                 then '_' + feature.name
331                 else feature.name
332                 endif,
333         dataType ← if (feature.oclIsKindOf(Ecore!EAttribute) and
    thisModule.enums→excludes(feature.eType))
334                     then  thisModule.dataType2simpleType(feature.eType)
335                     else feature.eType
336                     endif,
337         annotation ← memberAnnotation),
338         memberAnnotation: TDL!Annotation(
339             key ← if (thisModule.dynamicAnnotationType.oclIsUndefined())
340                     then thisModule.eAnnotation2annotationType()
341                     else thisModule.dynamicAnnotationType
342                     endif,
343             annotatedElement ← member)
344 }
345 unique lazy rule eAnnotation2annotationType{
346     from feature: Ecore!EStructuralFeature
347     to annotation: TDL!AnnotationType(
348         name ← 'dynamic')
349     do{
350         thisModule.dynamicAnnotationType ← annotation;
351     }
352 }
353 unique lazy rule abstract2annotationType{
354     from class: Ecore!EClass
355     to annotation: TDL!AnnotationType(
```

```
356         name ← 'abstract')
357     do{
358         thisModule.abstractAnnotationType ← annotation;
359     }
360 }
361 rule enum2simpleType{
362     from enum: Ecore!EEnum
363     to type : TDL!SimpleDataType(
364         name ← if (thisModule.tokenNames.includes(enum.name))
365                 then '_' + enum.name
366                 else enum.name
367                 endif)
368     do{
369         thisModule.tokenNames ← thisModule.tokenNames.append(type.name.
    toString());
370     }
371 }
372 rule enumLiteral2simpleDataInstance{
373     from enumLiteral: Ecore!EEnumLiteral
374     to dataInstance: TDL!SimpleDataInstance(
375         name ← if (thisModule.tokenNames.includes(enumLiteral.name))
376                 then '_' + enumLiteral.name
377                 else enumLiteral.name
378                 endif,
379         dataType ← enumLiteral.eEnum)
380     do{
381         thisModule.tokenNames ← thisModule.tokenNames.append(
    dataInstance.name.toString());
382     }
383 }
384 unique lazy rule dataType2simpleType{
385     from dataType: Ecore!EDataType
386     to type : TDL!SimpleDataType(
387         name ← if (thisModule.tokenNames.includes(dataType.name))
388                 then '_' + dataType.name
389                 else dataType.name
390                 endif)
391     do{
392         thisModule.tokenNames ← thisModule.tokenNames.append(type.name.
    toString());
393     }
394 }
395 rule tdlPackage{
396     from package: Ecore!EPackage
397     to dslTypesPackage: TDL!Package(
398         name ← package.name.concat('SpecificTypes'),
399         packagedElement ← package.eClassifiers.
400                           union(thisModule.enums).
401                           union(thisModule.enumLiterals).
```

```
402                                 append(thisModule.dynamicAnnotationType).
403                                 append(thisModule.abstractAnnotationType).
404                                 union(package.basicDataTypes→collect(t |
    thisModule.dataType2simpleType(t))))
405 }
```

Listing A.1: Ecore to TDL transformation rules implemented in ATL

Appendix B

Example 2: xPSSM

This appendix shows the result of using the proposed testing framework for another xDSL which is different from the xArduino.

B.1 Running Example 2: PSSM

UML State Machines is a well-known subset of the Unified Modeling Language (UML) standard [116] commonly used to model systems with discrete event-driven behavior. The Precise Semantics of UML State Machines (PSSM) is a standardized extension of UML that defines a complete execution semantics for UML State Machines [114]. Here, we rely on a simplified version of PSSM as our second running example, referred to as xPSSM. xPSSM only contains elements related to the reactive behavior of UML State Machines. Essentially, an xPSSM model is a state machine that can process external occurrences of events and perform behaviors in reaction. Figure B.1 shows an overview of different parts of the xPSSM language definition, and we present each part in the reminder of this section.

B.2 xPSSM Abstract Syntax

Figure B.1(**a**) briefly shows the abstract syntax of xPSSM defined as a metamodel. The root element is a CustomSystem. It contains one StateMachine and can have several Signals which will be used in its StateMachine. A StateMachine comprises one or more Region, each represents a behavior fragment that may execute concurrently with other regions if they are owned by either the same State or StateMachine. A Region is a graph comprising a set of Vertices interconnected by Transitions, which determines the behavioral flow within the Region.

Pseudostate and State are two kinds of Vertex. Pseudostates are *transitive*, meaning that the execution passes through them without pause. There are different

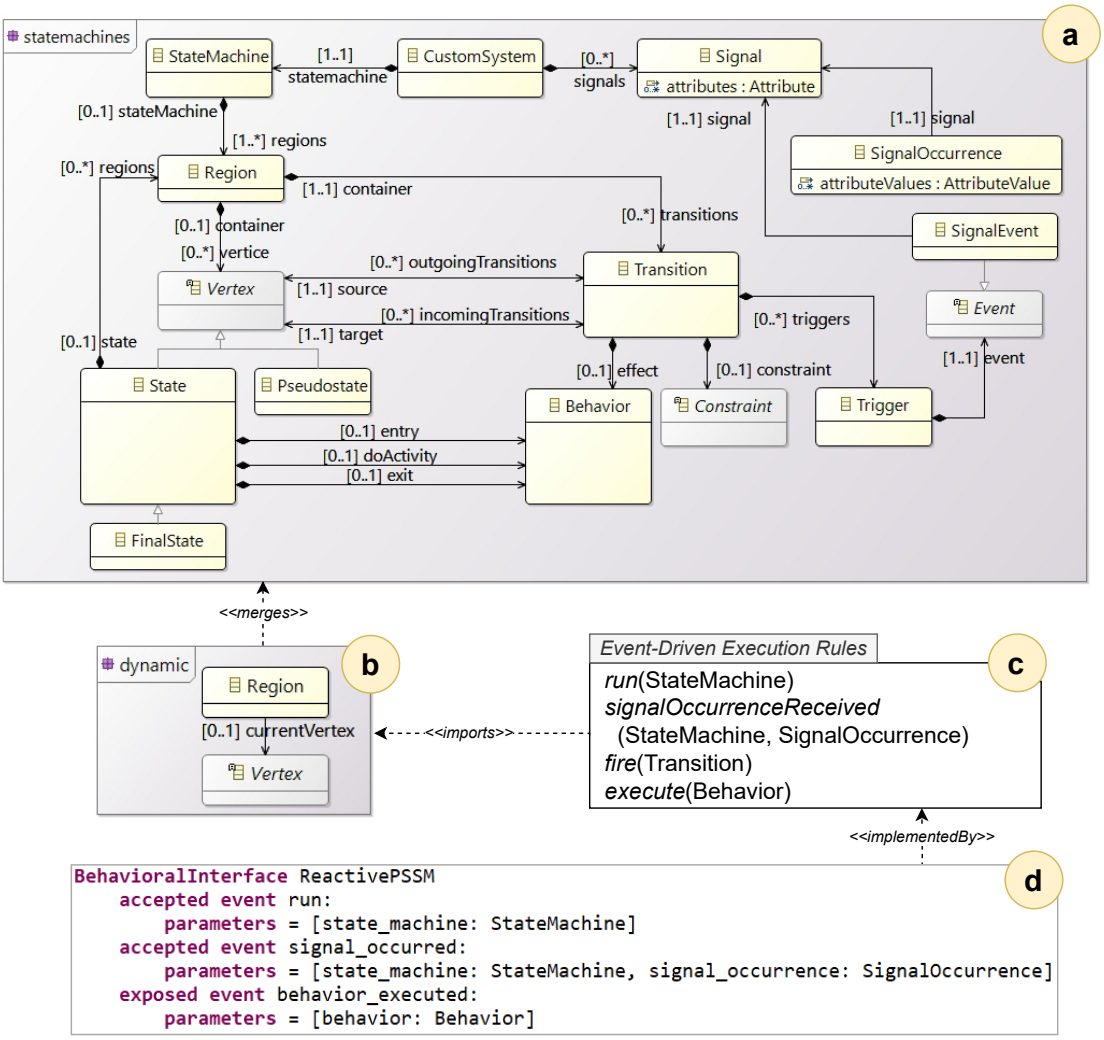


Figure B.1: A reactive xDSL for a subset of UML State Machines conforming to the PSSM specification [114] (referred to as xPSSM).

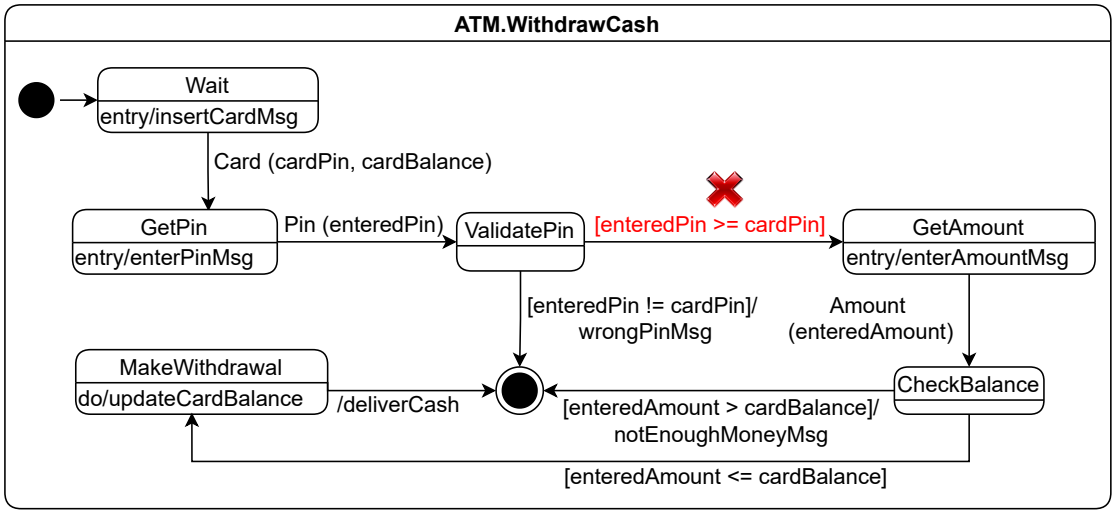


Figure B.2: A sample xPSSM model for cash withdrawal from an ATM. It has a defect since it does not validate the entered pin correctly (the wrong constraint is highlighted in red where >= is used instead of ==)

kinds of Pseudostates such as initial, fork, join, terminate. In contrast, State is a *stable* vertex which means when the execution enters them, it leaves when either some event occurs that triggers a Transition moving to another State or the StateMachine is terminated. A State may have `entry`, `doActivity`, and `exit` Behaviors—in our simplified xPSSM, a Behavior is an empty element without any substance. The `entry` and `exit` behaviors are executed when the State is entered and exited, respectively. Execution of the `doActivity` behavior starts after the `entry` Behavior (if any) has completed, and finishes when either it is completed or the State is exited. FinalState is a special kind of State representing the completion of its Region container.

A Transition connects a source vertex to a target one. It can contain three main elements: a Constraint, a Behavior, and several Triggers. A Transition is enabled when its guard Constraint (if any) evaluates to *true*, and its Behavior (if any) is executed once the transition is traversed. The traversal of the transitions may depend on the reception of the event occurrences. This is defined by allocating Trigger elements to them. A Trigger specifies an Event such as SignalEvent whose occurrence (i. e., SignalOccurrence) enables the traversal of the transition containing the Trigger. The SignalOccurrence contains *values* for the *attributes* of its associated Signal. When a state machine receives a signal occurrence, all the enabled transitions that contain a Trigger pointing to the related Signal will be traversed.

Figure B.2 shows an example model conforming to xPSSM. It describes a StateMachine that models the behavior of withdrawing cash from an Automated Teller

Machine (ATM). The bank `Card`, the entered `Pin` and the `Amount` of withdrawal are Signals whose specific occurrences can be given to the state machine at runtime using SignalOccurrences. The `Card` Signal has two attributes for its pin and balance. This StateMachine has one Region comprising one initial Pseudostate, one FinalState, seven States that three of them have entry Behavior (such as the `insertCardMsg` of the `Wait` State) and one of them has a doActivity Behavior (the `updateCardBalance` of the `MakeWithdrawal` State), and several Transitions which some require signal occurrences to get enabled. For example, the transition from `Wait` to `GetPin` state will be enabled once the state machine receives a SignalOccurrence for the `Card` Signal. Also, the outgoing transition of the `MakeWithdrawal` state has a behavior, namely `deliverCash`.

There are two conditions for a successful withdrawal. First, the entered `Pin` must be equals to the `Card's pin`. It is defined as a Constraint for the outgoing transition of the `ValidatePin` state, but with a wrong operator (highlighted in red in Figure B.2). Second, the entered `Amount` must be lower than equals to the `Card's balance` (i. e., the Constraint of the outgoing transition of the `CheckBalance` state).

B.3 Event-Driven Semantics of xPSSM

Figure B.1(**d**) shows a behavioral interface for xPSSM containing three event definitions, implemented by the execution rules listed in part (c):

- accepted event *run*: triggers the initialization of its state machine parameter (implemented by run(StateMachine)).
- accepted event *signal_occurred*: takes a signal occurrence as parameter and triggers its corresponding execution steps in the state machine (implemented by signalOccurrenceReceived (StateMachine, SignalOccurrence)).
- exposed event *behavior_executed*: notifies the execution of the Behavior elements (implemented by execute(Behavior)).

As an example, to execute the *ATM.WithdrawCash* state machine (Figure B.2), event occurrences conforming to xPSSM's behavioral interface should be communicated to the state machine. One can first send a run event with the *ATM.WithdrawCash* state machine as its parameter. This starts the execution and resulted in activating the `Wait` state and executing its `insertCardMsg` entry behavior which will be exposed by the model through a behavior_executed event. It is indeed the state machine reaction to receiving the run event occurrence. As the `currentVertex` is the `Wait` state, an occurrence for the signal_occurred event must be sent to the state machine with a `Card` instance to pursue.

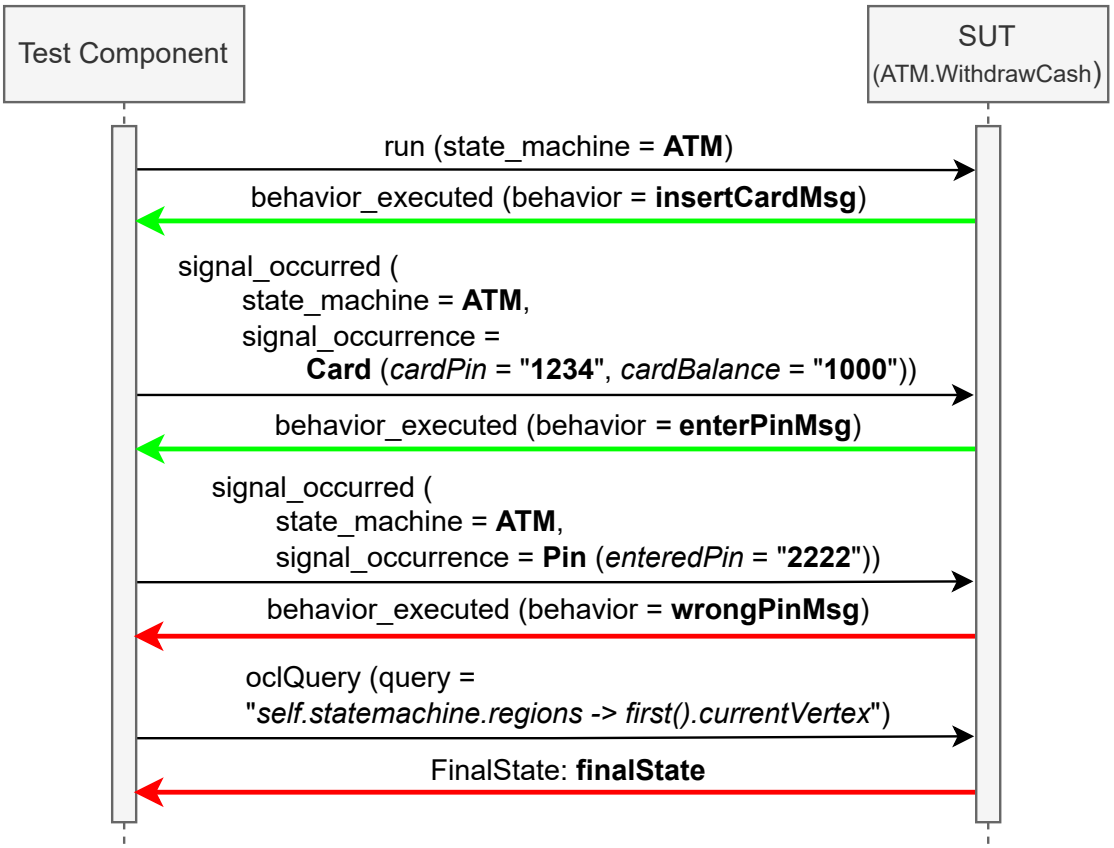


Figure B.3: A potential event-driven TDL test case for the xslePSSM running example of Figure B.2, with two passed and two failed assertions.

An Event-Driven Test Case for the xPSSM Example Model Figure B.3 shows an event-driven test case for the *ATM.WithdrawCash* state machine (previously shown in Figure B.2). The test case aims to check that the ATM does not accept an incorrect pin code, so it must be able to uncover the defect of the model. As can be seen, the events used in the test case conform to the xPSSM's behavioral interface (Figure B.1(**d**)) and their parameters are references to the elements of the ATM state machine.

First, the test component sends a *run* event to request the start of the execution and expects to receive in return a *behavior_executed* event for the `insertCardMsg` behavior. This assertion passes (the first green arrow in Figure B.3) because according to Figure B.2, when the state machine initializes, the execution should enter the `Wait` state, execute its `entry` behavior named `insertCardMsg`, and wait there until one of its outgoing transitions can be traversed.

Next, the test component sends a *signal_occurred* event with an occurrence

of the `Card` signal and expects to receive in return a *behavior_executed* event for the `enterPinMsg` behavior. As the state machine execution is currently in the `Wait` state, by receiving this event from the test component, the transition to the `GetPin` state will be traversed. So the execution enters this state and runs its `entry` behavior named `enterPinMsg`. Therefore, the second assertion also succeeds (the second green arrow in Figure B.3).

Afterward, the test component sends another *signal_occurred* event with an occurrence of the `Pin` signal and since the value of the entered pin (i. e., 2222) is different from the the card's pin (i. e., 1234), it expects to receive a *behavior_executed* event for the `wrongPinMsg` behavior. According to Figure B.2, as the state machine execution is currently in the `GetPin` state, receiving this event from the test component resulted in traversing the transition to the `ValidatePin` state. At this point, the constraints of its outgoing transitions are evaluated to check if they are enabled. However, as explained earlier, the ATM state machine contains a defect: an equality sign was mistakenly replaced by a superior-or-equal sign, leading to the wrong constraint "enteredPin >= cardPin". Consequently, instead of enabling the transition to the `finalState`, the one to the `GetAmount` state is enabled. Therefore, the `wrongPinMsg` event is never observed, meaning that the third assertion of the test case fails (the first red arrow in Figure B.3).

Finally, the test component sends an OCL query to check whether the `currentVertex` is the `finalState`. As described above, due to the defect of the model, the execution is currently in the `GetAmount` state, so the assertion fails (the second red arrow in Figure B.3).

B.4 xPSSM-Specific TDL Library

In this section, we present the content of the PSSM-specific TDL library produced by our proposed testing framework (in particular, by its *TDL Library Generator* component).

xPSSM-Specific Types Package: Listing B.1 presents an excerpt of the TDL DataTypes generated from the Ecore metamodel of the xPSSM (Figure B.1(**a**)).

```
1  Package xPSSMTypes {
2      Type CustomSystem (
3          statemachine of type StateMachine,
4          signals of type Signal);
5      Type Signal (
6          attributes of type Attribute) ;
7      Type SignalOccurrence (
8          signal of type Signal,
9          attributeValues of type AttributeValue);
10     Type StateMachine (
11         _name of type EString,
```

```
        regions of type Region);
    Type Behavior(
        _name of type EString);
    ...
}
```

Listing B.1: Some of the TDL Data Types generated for the xPSSM DSL

xPSSM-Specific Events Package: Listing B.2 shows the *xDSL-Specific Events Package* generated for the behavioral interface of the xPSSM (Figure B.1(**d**)).

```
Package xPSSMEvents {
    Import all from xPSSMTypes;

    Annotation AcceptedEvent;
    Annotation ExposedEvent;

    Type run (state_machine of type StateMachine)
        with{AcceptedEvent;};
    Type signal_occurred (
        state_machine of type StateMachine,
        signal_occurrence of type SignalOccurrence
    ) with {AcceptedEvent;};
    Type behavior_executed (behavior of type Behavior)
        with {ExposedEvent;};
}
```

Listing B.2: TDL elements generated for the xPSSM behavioral interface

Test Configuration Package: Listing B.3 demonstrates the test configuration generated for the xPSSM.

```
Package testConfiguration {
    Import all from common;
    Import all from xPSSMEvents;

    Gate Type genericGateType accepts modelExecutionCommand;
    Gate Type oclGateType accepts OCL;
    Gate Type reactiveGateType accepts run , signal_occurred ,
    behavior_executed;
    Component Type component having {
        gate genericGate of type genericGateType;
        gate oclGate of type oclGateType;
        gate reactiveGate of type reactiveGateType;
    }
    Annotation MUTPath;
    Annotation DSLName;

    Test Configuration xPSSMConfiguration {
        create Tester tester of type component;
        create SUT statemachine of type component with {
```

```
            MUTPath: 'TODO : Put the path to the MUT';
            DSLName: 'ReactivePSSM';
        };
        connect tester.genericGate to statemachine.genericGate;
        connect tester.oclGate to statemachine.oclGate;
        connect tester.reactiveGate to statemachine.reactiveGate;
    }
}
```

Listing B.3: TDL test configuration package generated for the xPSSM DSL

Using the TDL Library to write Test Cases: For example, we used the generated TDL library for the xPSSM to write an executable TDL test case for the ATM sample model, presented in Listing B.4.

```
Package reactiveATM_testSuite {
    Import all from common;
    Import all from xPSSMTypes;
    Import all from xPSSMEvents;
    Import all from testConfiguration;

    StateMachine ATM (_name = "ATM.WithdrawCash");
    Behavior insertCardMsg (_name = "insertCardMsg");

    Test Description test_wrongPin uses configuration xPSSMConfiguration{
        tester.reactiveGate sends run (state_machine = ATM)
            to statemachine.reactiveGate;
        statemachine.reactiveGate sends behavior_executed (
            behavior = insertCardMsg) to tester.reactiveGate;
        tester.reactiveGate sends signal_occurred (
            state_machine = ATM,
            signal_occurrence = card_occurrence (
                signal = Card,
                attributeValues = {cardPinValue (value = "1234"),
                cardBalanceValue (value = "1000")})
            ) to statemachine.reactiveGate;
        statemachine.reactiveGate sends behavior_executed (
            behavior = enterPinMsg) to tester.reactiveGate;
        tester.reactiveGate sends signal_occurred (
            state_machine = ATM,
            signal_occurrence = pin_occurrence (
                signal = Pin,
                attributeValues = {enteredPinValue (value = "2222")})
            ) to statemachine.reactiveGate;
        statemachine.reactiveGate sends behavior_executed (
            behavior = wrongPinMsg) to tester.reactiveGate;
        tester.oclGate sends oclQuery (
            query =  "self.statemachine.regions->first().currentVertex")
            to statemachine.oclGate;
        statemachine.oclGate sends finalState to tester.oclGate;
```

```
36      }
37  }
```

Listing B.4: An event-driven TDL test case for testing the ATM withdraw cash xPSSM model.

List of Figures

List of Tables

Bibliography

[1] M. Abdi, H. Rocha, S. Demeyer, and A. Bergel. "Small-Amp: Test amplification in a dynamically typed language". In: *Empirical Software Engineering* 27.6 (2019), p. 128. DOI: 10.1007/s10664-022-10169-8.

[2] R. Abreu, P. Zoeteweij, and A. J. van Gemund. "Spectrum-Based Multiple Fault Localization". In: *2009 IEEE/ACM International Conference on Automated Software Engineering*. 2009, pp. 88–99. DOI: 10.1109/ASE.2009.25.

[3] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. van Gemund. "A practical evaluation of spectrum-based fault localization". In: *Journal of Systems and Software* 82.11 (2009), pp. 1780–1792. ISSN: 0164-1212. DOI: 10.1016/j.jss.2009.06.035.

[4] H. Agrawal, R. A. DeMillo, R. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford. *Design of mutant operators for the C programming language*. Tech. rep. Purdue University, 1989.

[5] T. Ahmad, J. Iqbal, A. Ashraf, D. Truscan, and I. Porres. "Model-based testing using UML activity diagrams: A systematic mapping study". In: *Computer Science Review* 33 (2019), pp. 98–112. DOI: 10.1016/j.cosrev.2019.07.001.

[6] R. Ahmadi, N. Hili, and J. Dingel. "Property-Aware Unit Testing of UML-RT Models in the Context of MDE". In: *Proceedings of the 14th European Conference on Modelling Foundations and Applications (ECMFA)*. Vol. 10890. Lecture Notes in Computer Science. Springer, 2018, pp. 147–163. DOI: 10.1007/978-3-319-92997-2_10.

[7] B. K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, and S. Tiran. "Killing strategies for model-based mutation testing". In: *Software Testing, Verification and Reliability* 25.8 (2015), pp. 716–748. DOI: 10.1002/stvr.1522.

[8] I. Al-Azzoni and S. Iqbal. "A Framework for the Regression Testing of Model-to-Model Transformations". In: *e-Informatica Software Engineering Journal* 15.1 (2021), pp. 65–84. DOI: 10.37190/e-Inf210104.

[9] F. H. M. Alhwikem, R. F. Paige, L. M. Rose, and R. D. Alexander. "A systematic approach for designing mutation operators for MDE languages". In: *Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa)*. 2016, pp. 54–59. URL: https://eprints.whiterose.ac.uk/106909/.

[10] B. Alkhazi, C. Abid, M. Kessentini, D. Leroy, and M. Wimmer. "Multi-Criteria Test Cases Selection for Model Transformations". In: *Automated Software Engineering* 27.1–2 (2020), 91–118. ISSN: 0928-8910. DOI: 10.1007/s10515-020-00271-w.

[11] S. C. Allala, J. P. Sotomayor, D. Santiago, T. M. King, and P. J. Clarke. "Towards Transforming User Requirements to Test Cases Using MDE and NLP". In: *43rd IEEE Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, 2019, pp. 350–355. DOI: 10.1109/COMPSAC.2019.10231.

[12] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[13] J. Andrews, L. Briand, Y. Labiche, and A. Namin. "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria". In: *IEEE Transactions on Software Engineering* 32.8 (2006), pp. 608–624. DOI: 10.1109/TSE.2006.83.

[14] V. Aranega, J.-M. Mottu, A. Etien, T. Degueule, B. Baudry, and J.-L. Dekeyser. "Towards an automation of the mutation analysis dedicated to model transformation". In: *Software Testing, Verification and Reliability* 25.5-7 (2015), pp. 653–683.

[15] L. Arcega, J. Font, and C. Cetina. "Evolutionary algorithm for bug localization in the reconfigurations of models at runtime". In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. 2018, pp. 90–100. DOI: 10.1145/3239372.3239392.

[16] L. Arcega, J. Font, Ø. Haugen, and C. Cetina. "An approach for bug localization in models using two levels: model and metamodel". In: *Software and Systems Modeling* 18.6 (2019), pp. 3551–3576. DOI: 10.1007/s10270-019-00727-y.

[17] L. Arcega, J. Font, Ø. Haugen, and C. Cetina. "Bug Localization in Model-Based Systems in the Wild". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31.1 (2021), pp. 1–32. DOI: 10.1145/3472616.

[18] M. Arnaud, B. Bannour, A. Cuccuru, C. Gaston, S. Gerard, and A. Lapitre. "Timed symbolic testing framework for executable models using high-level scenarios". In: *Complex Systems Design & Management*. Springer, 2015, pp. 269–282. DOI: 10.1007/978-3-319-11617-4_19.

[19] F. Y. Assiri and J. M. Bieman. "Fault localization for automated program repair: effectiveness, performance, repair correctness". In: *Software Quality Journal* 25.1 (2017), pp. 171–199. ISSN: 1573-1367. DOI: 10.1007/s11219-016-9312-z.

[20] T. B. de Assis, A. A. Menegassi, and A. T. Endo. "Amplifying Tests for Cross-Platform Apps through Test Patterns". In: *The 31st International Conference on Software Engineering and Knowledge Engineering, SEKE*. Ed. by A. Perkusich. KSI Research Inc. and Knowledge Systems Institute Graduate School, 2019, pp. 55–74. DOI: 10.18293/SEKE2019-076.

[21] Atlanmod. *xArduino with Kermeta semantics*. 2020. URL: https://github.com/atlanmod/eel/tree/master/Language_Workbench.

[22] N. Bandener, C. Soltenborn, and G. Engels. "Extending DMM Behavior Specifications for Visual Execution and Debugging". In: *Software Language Engineering*. Ed. by B. Malloy, S. Staab, and M. van den Brand. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 357–376. DOI: 10.1007/978-3-642-19440-5_24.

[23] A. Bauer, M. Leucker, and C. Schallhart. "Runtime Verification for LTL and TLTL". In: *ACM Trans. Softw. Eng. Methodol.* 20.4 (2011). ISSN: 1049-331X. DOI: 10.1145/2000799.2000800.

[24] R. Bendraou, B. Combemale, X. Cregut, and M.-P. Gervais. "Definition of an Executable SPEM 2.0". In: *14th Asia-Pacific Software Engineering Conference (APSEC'07)*. 2007, pp. 390–397. DOI: 10.1109/ASPEC.2007.60.

[25] E. Biermann, C. Ermel, and G. Taentzer. "Precise Semantics of EMF Model Transformations by Graph Transformation". In: *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems*. MoDELS '08. Toulouse, France: Springer-Verlag, 2008, 53–67. ISBN: 9783540878742. DOI: 10.1007/978-3-540-87875-9_4.

[26] R. Bill, M. Fleck, J. Troya, T. Mayerhofer, and M. Wimmer. "A local and global tour on MOMoT". In: *Software and Systems Modeling* 18.2 (2019), pp. 1017–1046. DOI: 10.1007/s10270-017-0644-3.

[27] P. N. Boghdady, N. L. Badr, M. A. Hashim, and M. F. Tolba. "An enhanced test case generation technique based on activity diagrams". In: *The 2011 International Conference on Computer Engineering & Systems*. IEEE. 2011, pp. 289–294. DOI: 10.1109/ICCES.2011.6141058.

[28] M. Bordin, C. Comar, T. Gingold, J. Guitton, O. Hainque, T. Quinot, J. Delange, J. Hugues, and L. Pautet. "Couverture: an innovative open framework for coverage analysis of safety critical applications". In: *Ada User Journal* 30.4 (2009), pp. 248–255.

[29] E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. Deantoni, and B. Combemale. "Execution Framework of the GEMOC Studio (Tool Demo)". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. Association for Computing Machinery, 2016, 84–89. DOI: 10.1145/2997364.2997384.

[30] E. Bousse, D. Leroy, B. Combemale, M. Wimmer, and B. Baudry. "Omniscient debugging for executable DSLs". In: *Journal of Systems and Software* 137 (2018), pp. 261–288. ISSN: 0164-1212. DOI: 10.1016/j.jss.2017.11.025.

[31] E. Bousse, T. Mayerhofer, B. Combemale, and B. Baudry. "A Generative Approach to Define Rich Domain-Specific Trace Metamodels". In: *Modelling Foundations and Applications*. Ed. by G. Taentzer and F. Bordeleau. Cham: Springer International Publishing, 2015, pp. 45–61. ISBN: 978-3-319-21151-0. DOI: 10.1007/978-3-319-21151-0_4.

[32] E. Bousse, T. Mayerhofer, B. Combemale, and B. Baudry. "Advanced and efficient execution trace management for executable domain-specific modeling languages". In: *Software and Systems Modeling* (Feb. 2019), pp. 1–37. DOI: 10.1007/s10270-017-0598-5.

[33] E. Bousse and M. Wimmer. "Domain-Level Observation and Control for Compiled Executable DSLs". In: *IEEE / ACM 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Munich, Germany, Sept. 2019. DOI: 10.1109/MODELS.2019.000-6.

[34] J. S. Bradbury, J. R. Cordy, and J. Dingel. "Mutation Operators for Concurrent Java (J2SE 5.0)". In: *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)*. 2006, pp. 11–11. DOI: 10.1109/MUTATION.2006.10.

[35] M. Brambilla, J. Cabot, and M. Wimmer. "Model-driven software engineering in practice". In: *Synthesis lectures on software engineering* 3.1 (2017), pp. 1–207. DOI: 10.1007/978-3-031-02549-5.

[36] P. C. Cañizares, P. Gómez-Abajo, A. Núñez, E. Guerra, and J. de Lara. "New ideas: automated engineering of metamorphic testing environments for domain-specific languages". In: *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2021. Chicago, IL, USA: ACM, 2021, pp. 49–54. DOI: 10.1145/3486608.3486904.

[37] F. Ciccozzi, I. Malavolta, and B. Selic. "Execution of UML models: a systematic review of research and practice". In: *Software and Systems Modeling* 18 (2019), 2313–2360. DOI: 10.1007/s10270-018-0675-4.

[38] Cobertura. *A code coverage utility for Java*. 2022. URL: http://cobertura.github.io/cobertura/.

[39] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque. "PIT: A Practical Mutation Testing Tool for Java (Demo)". In: *International Symposium on Software Testing and Analysis (ISSTA)*. See also https://pitest.org/quickstart/mutators, https://github.com/hcoles/pitest. Saarbrücken, Germany: ACM, 2016, pp. 449–452. ISBN: 978-1-4503-4390-9. DOI: 10.1145/2931037.2948707.

[40] G. Csertan, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varro. "VIATRA - visual automated transformations for formal verification and validation of UML models". In: *Proceedings 17th IEEE International Conference on Automated Software Engineering,* 2002, pp. 267–270. DOI: 10.1109/ASE.2002.1115027.

[41] C. Damus, A. Sánchez-Barbudo Herrera, A. Uhl, and E. Willink. *OCL Documentation*. Tech. rep. 2021, p. 227. URL: http://download.eclipse.org/ocl/doc/6.15.0/ocl.pdf.

[42] B. Danglot, O. Vera-Perez, Z. Yu, A. Zaidman, M. Monperrus, and B. Baudry. "A snowballing literature study on test amplification". In: *Journal of Systems and Software* 157 (2019), p. 110398. ISSN: 0164-1212. DOI: 10.1016/j.jss.2019.110398.

[43] B. Danglot, O. L. Vera-Pérez, B. Baudry, and M. Monperrus. "Automatic Test Improvement with DSpot: a Study with Ten Mature Open-Source Projects". In: *Empirical Software Engineering* 24.4 (2019), pp. 1–35. DOI: 10.1007/s10664-019-09692-y.

[44] J. Deantoni. "Modeling the Behavioral Semantics of Heterogeneous Languages and their Coordination". In: *Architecture Centric Virtual Integration (ACVI)*. Julien Delange and Jerome Hugues and Peter Feiler. 2016. DOI: 10.1109/ACVI.2016.9.

[45] J. Deantoni, I. P. Diallo, C. Teodorov, J. Champeau, and B. Combemale. "Towards a meta-language for the concurrency concern in DSLs". In: *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2015, pp. 313–316. DOI: 10.7873/DATE.2015.1052.

[46] P. Delgado-Pérez, I. Medina-Bulo, F. Palomo-Lozano, A. García-Domínguez, and J. J. Domínguez-Jiménez. "Assessment of class mutation operators for C++ with the MuCPP mutation system". In: *Information and Software Technology* 81 (2017), pp. 169–184. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2016.07.002.

[47] R. DeMillo, R. Lipton, and F. Sayward. "Hints on Test Data Selection: Help for the Practicing Programmer". In: *Computer* 11.4 (1978), pp. 34–41. DOI: 10.1109/C-M.1978.218136.

[48] D. Di Ruscio, D. Kolovos, J. de Lara, A. Pierantonio, M. Tisi, and M. Wimmer. "Low-code development and model-driven engineering: Two sides of the same coin?" In: *Software and Systems Modeling* 21.2 (2022), pp. 437–446. DOI: 10.1007/s10270-021-00970-2.

[49] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, and M. Hanus. "Xbase: Implementing Domain-Specific Languages for Java". In: *SIGPLAN Notices* 48.3 (2012), 112–121. DOI: 10.1145/2480361.2371419.

[50] M. El qortobi, A. Rahj, J. Bentahar, and R. Dssouli. "Test Generation Tool for Modified Condition/Decision Coverage: Model Based Testing". In: *Proceedings of the 13th International Conference on Intelligent Systems: Theories and Applications*. 2020, pp. 1–6. DOI: 10.1145/3419604.3419628.

[51] S. Erdweg, T. v. d. van der Storm, M. Voelter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. v. d. van der Vlist, G. Wachsmuth, and J. v. d. van der Woning. "Evaluating And Comparing Language Workbenches: Existing Results And Benchmarks For The Future". In: *Computer Languages, Systems and Structures* 44.Part A (2015), pp. 24 –47. DOI: 10.1016/j.cl.2015.08.007.

[52] ETSI ES 203 119-1. *Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 1: abstract syntax and associated semantics*. Version V1.6.1. Sophia-Antipolis, France: European Telecommunications Standards Institute (ETSI), 2022. URL: https://tdl.etsi.org/index.php/downloads.

[53] ETSI ES 203 119-6. *Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 6: Mapping to TTCN-3*. Version V1.2.1. Sophia-Antipolis, France: European Telecommunications Standards Institute (ETSI), 2020. URL: https://tdl.etsi.org/index.php/downloads.

[54] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, M. E. Delamaro, and E. Wong. "Mutation testing applied to validate specifications based on Petri Nets". In: *Formal Description Techniques VIII: Proceedings of the IFIP TC6 Eighth International Conference on Formal Description Techniques, Montreal, Canada, October 1995*. Ed. by G. v. Bochmann, R. Dssouli, and O. Rafiq. Boston, MA: Springer US, 1996, pp. 329–337. ISBN: 978-0-387-34945-9. DOI: 10.1007/978-0-387-34945-9_24.

[55] S. Fabbri, J. Maldonado, and M. Delamaro. "Proteum/FSM: a tool to support finite state machine validation based on mutation testing". In: *Proceedings. SCCC'99 XIX International Conference of the Chilean Computer Science Society*. 1999, pp. 96–104. DOI: 10.1109/SCCC.1999.810159.

[56] M. Fowler. *Domain-specific languages*. Pearson Education, 2010. URL: https://martinfowler.com/books/dsl.html.

[57] R. France and B. Rumpe. "Model-driven Development of Complex Software: A Research Roadmap". In: *Future of Software Engineering (FOSE '07)*. 2007, pp. 37–54. DOI: 10.1109/FOSE.2007.14.

[58] P. Fröhlich and J. Link. "Automated Test Case Generation from Dynamic Models". In: *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP)*. Ed. by E. Bertino. Vol. 1850. Lecture Notes in Computer Science. Springer, 2000, pp. 472–492. DOI: 10.1007/3-540-45102-1_23.

[59] A. A. Giron, I. M. de Souza Gimenes, and E. OliveiraJr. "Evaluation of Test Case Generation based on a Software Product Line for Model Transformation". In: *Journal of Computer Science* 14.1 (2018), pp. 108–121. DOI: 10.3844/jcssp.2018.108.121.

[60] P. Gómez-Abajo, E. Guerra, J. de Lara, and M. G. Merayo. "A tool for domain-independent model mutation". In: *Science of Computer Programming* 163 (2018), pp. 85–92. ISSN: 0167-6423. DOI: 10.1016/j.scico.2018.01.008.

[61] P. Gómez-Abajo, E. Guerra, and J. de Lara. "Wodel: A Domain-Specific Language for Model Mutation". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. SAC '16. Association for Computing Machinery, 2016, 1968–1973. DOI: 10.1145/2851613.2851751.

[62] P. Gómez-Abajo, E. Guerra, J. de Lara, and M. G. Merayo. "Wodel-Test: a model-based framework for language-independent mutation testing". In: *Software and Systems Modeling* 20 (2020), pp. 1–27. DOI: 10.1007/s10270-020-00827-0.

[63] J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock. "An introduction to the testing and test control notation (TTCN-3)". In: *Computer Networks* 42.3 (2003). ITU-T System Design Languages (SDL), pp. 375 –403. ISSN: 1389-1286. DOI: 10.1016/S1389-1286(03)00249-4.

[64] M. F. Granda, N. Condori-Fernández, T. E. J. Vos, and O. Pastor. "Mutation Operators for UML Class Diagrams". In: *Advanced Information Systems Engineering*. Ed. by S. Nurcan, P. Soffer, M. Bajec, and J. Eder. Cham: Springer International Publishing, 2016, pp. 325–341. ISBN: 978-3-319-39696-5. DOI: 10.1007/978-3-319-39696-5_20.

[65] E. Guerra, J. S. Cuadrado, and J. de Lara. "Towards effective mutation testing for ATL". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE. 2019, pp. 78–88.

[66] E. Guerra and M. Soeken. "Specification-driven model transformation testing". In: *Software and Systems Modeling* 14.2 (2015), pp. 623–644. DOI: 10.1007/s10270-013-0369-x.

[67] R. G. Hamlet. "Testing Programs with the Aid of a Compiler". In: *IEEE Transactions on Software Engineering* 3.4 (1977), pp. 279–290. DOI: 10.1109/TSE.1977.231145.

[68] Á. Hegedüs, G. Bergmann, I. Z. Ráth, and D. Varró. "Back-annotation of Simulation Traces with Change-Driven Model Transformations". In: *8th IEEE International Conference on Software Engineering and Formal Methods (SEFM)*. 2010, pp. 145–155. DOI: 10.1109/SEFM.2010.28.

[69] Á. Hegedüs, I. Z. Ráth, and D. Varró. "Replaying execution trace models for dynamic modeling languages". In: (2013), 71–82. DOI: 10.3311/PPee.7078.

[70] A. R. Hevner, S. T. March, J. Park, and S. Ram. "Design science in information systems research". In: *MIS Quarterly* 28.1 (2004), pp. 75–105. DOI: 10.2307/25148625.

[71] N. Hili, M. Bagherzadeh, K. Jahed, and J. Dingel. "A model-based architecture for interactive run-time monitoring". In: *Software and Systems Modeling* 19 (2020), pp. 959–981. DOI: 10.1007/s10270-020-00780-y.

[72] F. Hojaji, T. Mayerhofer, B. Zamani, A. Hamou-Lhadj, and E. Bousse. "Model execution tracing: a systematic mapping study". In: *Software and Systems Modeling* 18.6 (Feb. 2019), pp. 3461–3485. DOI: 10.1007/s10270-019-00724-1.

[73] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. "Empirical assessment of MDE in industry". In: *Proceedings of the 33rd International Conference on Software Engineering*. ACM, May 2011. DOI: 10.1145/1985793.1985858.

[74] J. Iqbal, A. Ashraf, D. Truscan, and I. Porres. "Exhaustive Simulation and Test Generation Using fUML Activity Diagrams". In: *Advanced Information Systems Engineering*. Ed. by P. Giorgini and B. Weber. Cham: Springer International Publishing, 2019a, pp. 96–110. DOI: 10.1007/978-3-030-21290-2_7.

[75] M. Z. Iqbal, A. Arcuri, and L. Briand. "Environment Modeling and Simulation for Automated Testing of Soft Real-Time Embedded Software". In: *Software and Systems Modeling* 14.1 (2015), 483–524. ISSN: 1619-1366. DOI: 10.1007/s10270-013-0328-6.

[76] JaCoCo. *JaCoCo Java Code Coverage Library*. 2022. URL: https://github.com/jacoco/jacoco.

[77] T. Janssen, R. Abreu, and A. J. van Gemund. "Zoltar: A Spectrum-Based Fault Localization Tool". In: *Proceedings of the 2009 ESEC/FSE Workshop on Software Integration and Evolution @ Runtime*. SINTER '09. Amsterdam, The Netherlands: Association for Computing Machinery, 2009, 23–30. ISBN: 9781605586816. DOI: 10.1145/1596495.1596502.

[78] J.-M. Jézéquel, B. Combemale, O. Barais, M. Monperrus, and F. Fouquet. "Mashup of metalanguages and its implementation in the kermeta language workbench". In: *Software and Systems Modeling* 14.2 (2015), pp. 905–920. DOI: 10.1007/s10270-013-0354-4.

[79] Y. Jia and M. Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Transactions on Software Engineering* 37.5 (2011), pp. 649–678. DOI: 10.1109/TSE.2010.62.

[80] Y. Jia and M. Harman. "MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language". In: *Proceedings of the Testing: Academic Industrial Conference - Practice and Research Techniques*. TAIC-PART '08. USA: IEEE Computer Society, 2008, 94–98. ISBN: 9780769533834. DOI: 10.1109/TAIC-PART.2008.18.

[81] J. A. Jones and M. J. Harrold. "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique". In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ASE '05. Long Beach, CA, USA: Association for Computing Machinery, 2005, 273–282. ISBN: 1581139934. DOI: 10.1145/1101908.1101949.

[82] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. "ATL: A QVT-like Transformation Language". In: *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA '06. Portland, Oregon, USA, 2006, 719–720. DOI: 10.1145/1176617.1176691.

[83] T. Kehrer, G. Taentzer, M. Rindt, and U. Kelter. "Automatically Deriving the Specification of Model Editing Operations from Meta-Models". In: *Theory and Practice of Model Transformations*. Ed. by P. Van Gorp and G. Engels. Cham: Springer International Publishing, 2016, pp. 173–188. ISBN: 978-3-319-42064-6. DOI: 10.1007/978-3-319-42064-6_12.

[84] W. Kessentini, H. Sahraoui, and M. Wimmer. "Automated metamodel/-model co-evolution: A search-based approach". In: *Information and Software Technology* 106 (2019), pp. 49–67. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2018.09.003.

[85] S. Kim, J. A. Clark, and J. A. McDermid. "Investigating the Effectiveness of Object-Oriented Strategies with the Mutation Method". In: *Mutation Testing for the New Century*. Ed. by W. E. Wong. Boston, MA: Springer US, 2001, pp. 4–4. ISBN: 978-1-4757-5939-6. DOI: 10.1007/978-1-4757-5939-6_3.

[86] T. M. King, G. Nunez, D. Santiago, A. Cando, and C. Mack. "Legend: An Agile DSL Toolset for Web Acceptance Testing". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. Association for Computing Machinery, 2014, 409–412. DOI: 10.1145/2610384.2628048.

[87] T. Kos, M. Mernik, and T. Kosar. "Test automation of a measurement system using a domain-specific modelling language". In: *Journal of Systems and Software* 111 (2016), pp. 74 –88. DOI: 10.1016/j.jss.2015.09.002.

[88] T. Kosar, S. Bohra, and M. Mernik. "Domain-Specific Languages: A Systematic Mapping Study". In: *Information and Software Technology* 71 (Mar. 2016), pp. 77–91. DOI: 10.1016/j.infsof.2015.11.001.

[89] W. Krenn, R. Schlick, S. Tiran, B. Aichernig, E. Jobstl, and H. Brandl. "MoMut::UML Model-Based Mutation Testing for UML". In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 2015, pp. 1–8. DOI: 10.1109/ICST.2015.7102627.

[90] S. Kriebel, M. Markthaler, K. S. Salman, T. Greifenberg, S. Hillemacher, B. Rumpe, C. Schulze, A. Wortmann, P. Orth, and J. Richenhagen. "Improving model-based testing in automotive software engineering". In: *Proceedings of the 40th International Conference on Software Engineering: Software*

Engineering in Practice (ICSE-SEIP). ACM, 2018, pp. 172–180. DOI: 10.1145/3183519.3183533.

[91] I. Lazăr, S. Motogna, and B. Pârv. "Behaviour-Driven Development of Foundational UML Components". In: *Electronic Notes in Theoretical Computer Science* 264.1 (2010). Proceedings of the 7th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA 2010), pp. 91–105. DOI: 10.1016/j.entcs.2010.07.007.

[92] M. Leduc, T. Degueule, B. Combemale, T. van der Storm, and O. Barais. "Revisiting Visitors for Modular Extension of Executable DSMLs". In: *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2017, pp. 112–122. DOI: 10.1109/MODELS.2017.23.

[93] D. Leroy, E. Bousse, M. Wimmer, T. Mayerhofer, B. Combemale, and W. Schwinger. "Behavioral interfaces for executable DSLs". In: *Software and Systems Modeling* 19.4 (2020), pp. 1015–1043. DOI: 10.1007/s10270-020-00798-2.

[94] D. Leroy, P. Jeanjean, E. Bousse, M. Wimmer, and B. Combemale. "Runtime Monitoring for Executable DSLs". In: *The Journal of Object Technology* 19.2 (2020), pp. 1–23. DOI: 10.5381/jot.2020.19.2.a6.

[95] J.-h. Li, G.-x. Dai, and H.-h. Li. "Mutation Analysis for Testing Finite State Machines". In: *2009 Second International Symposium on Electronic Commerce and Security*. 2009, pp. 620–624. DOI: 10.1109/ISECS.2009.158.

[96] P. Li, M. Jiang, and Z. Ding. "Fault localization with weighted test model in model transformations". In: *IEEE Access* 8 (2020), pp. 14054–14064. DOI: 10.1109/ACCESS.2020.2966540.

[97] D. Lübke and T. van Lessen. "BPMN-Based Model-Driven Testing of Service-Based Processes". In: *Enterprise, Business-Process and Information Systems Modeling*. Cham: Springer, 2017, pp. 119–133. DOI: 10.1007/978-3-319-59466-8_8.

[98] P. Makedonski, G. Adamis, M. Käärik, F. Kristoffersen, M. Carignani, A. Ulrich, and J. Grabowski. "Test descriptions with ETSI TDL". In: *Software Quality Journal* 27.2 (2019), pp. 885–917. DOI: 10.1007/s11219-018-9423-9.

[99] X. Mao, Y. Lei, Z. Dai, Y. Qi, and C. Wang. "Slice-based statistical fault localization". In: *Journal of Systems and Software* 89 (2014), pp. 51–62. ISSN: 0164-1212. DOI: 10.1016/j.jss.2013.08.031.

[100] A. Maxwell and A. Pilliner. "Deriving coefficients of reliability and agreement for ratings". In: *British Journal of Mathematical and Statistical Psychology* 21.1 (1968), pp. 105–116. DOI: 10.1111/j.2044-8317.1968.tb00401.x.

[101] T. Mayerhofer and B. Combemale. "The Tool Generation Challenge for Executable Domain-Specific Modeling Languages". In: *Software Technologies: Applications and Foundations (STAF)*. Ed. by M. Seidl and S. Zschaler. Cham: Springer International Publishing, 2018, pp. 193–199. ISBN: 978-3-319-74730-9. DOI: 10.1007/978-3-319-74730-9_18.

[102] T. Mayerhofer, P. Langer, M. Wimmer, and G. Kappel. "xMOF: Executable DSMLs based on fUML". In: *International conference on software language engineering*. Springer. 2013, pp. 56–75. DOI: 10.1007/978-3-319-02654-1_4.

[103] T. Mens, A. Decan, and N. I. Spanoudakis. "A method for testing and validating executable statechart models". In: *Software and Systems Modeling* 18 (2 2019), pp. 837–863. DOI: 10.1007/s10270-018-0676-3.

[104] M. Mernik, J. Heering, and A. M. Sloane. "When and How to Develop Domain-Specific Languages". In: *ACM Comput. Surv.* 37.4 (2005), 316–344. ISSN: 0360-0300. DOI: 10.1145/1118890.1118892.

[105] B. Meyers, J. Denil, I. Dávid, and H. Vangheluwe. "Automated testing support for reactive domain-specific modelling languages". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. Association for Computing Machinery, 2016, pp. 181–194. DOI: 10.1145/2997364.2997367.

[106] S. Mijatov, T. Mayerhofer, P. Langer, and G. Kappel. "Testing Functional Requirements in UML Activity Diagrams". In: *Tests and Proofs (TAP)*. Ed. by J. C. Blanchette and N. Kosmatov. Cham: Springer International Publishing, 2015, pp. 173–190. DOI: 10.1007/978-3-319-21215-9_11.

[107] J. Misurda, J. A. Clause, J. L. Reed, B. R. Childers, and M. L. Soffa. "Demand-driven structural testing with dynamic instrumentation". In: *Proceedings of the 27th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2005, pp. 156–165. ISBN: 1581139632. DOI: 10.1145/1062455.1062496.

[108] M. L. Mohd-Shafie, W. M. N. W. Kadir, H. Lichter, M. Khatibsyarbini, and M. A. Isa. "Model-Based Test Case Generation and Prioritization: A Systematic Literature Review". In: *Software and Systems Modeling* 21.2 (2022), 717–753. ISSN: 1619-1366. DOI: 10.1007/s10270-021-00924-8.

[109] J.-M. Mottu, B. Baudry, and Y. L. Traon. "Mutation analysis testing for model transformations". In: *European Conference on Model Driven Architecture-Foundations and Applications*. Springer. 2006, pp. 376–390.

[110] L. Naish, H. J. Lee, and K. Ramamohanarao. "A Model for Spectra-Based Software Diagnosis". In: *ACM Trans. Softw. Eng. Methodol.* 20.3 (2011). ISSN: 1049-331X. DOI: 10.1145/2000791.2000795.

[111] OASIS. *Web Services Business Process Execution Language Version 2.0*. 2007. URL: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.

[112] Object Management Group. *Business Process Model And Notation*. Version 2.0. 2010. URL: https://www.bpmn.org/.

[113] Object Management Group. *Meta Object Facility*. Version 2.5.1. 2016. URL: https://www.omg.org/spec/MOF.

[114] Object Management Group. *Precise Semantics of UML State Machines*. Version 1.0. 2019. URL: https://www.omg.org/spec/PSSM/1.0/.

[115] Object Management Group. *Semantics of a Foundational Subset for Executable UML Models*. Version 1.5. 2013. URL: https://www.omg.org/spec/FUML/1.5/.

[116] Object Management Group. *Unified Modeling Language*. Version 2.5.1. 2017. URL: https://www.omg.org/spec/UML/2.5.1/.

[117] A. H. Patil and N. S. Sidnal. *CodeCover: A Code Coverage Tool for Java Projects*. 2013.

[118] K. Petersen, S. Vakkalanka, and L. Kuzniarz. "Guidelines for conducting systematic mapping studies in software engineering: An update". In: *Information and Software Technology* 64 (2015), pp. 1–18. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2015.03.007.

[119] S. Pinto Ferraz Fabbri, M. Delamaro, J. Maldonado, and P. Masiero. "Mutation analysis testing for finite state machines". In: *Proceedings of 1994 IEEE International Symposium on Software Reliability Engineering*. 1994, pp. 220–229. DOI: 10.1109/ISSRE.1994.341378.

[120] Y. Qi, X. Mao, Y. Lei, and C. Wang. "Using Automated Program Repair for Evaluating the Effectiveness of Fault Localization Techniques". In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ISSTA 2013. Lugano, Switzerland: Association for Computing Machinery, 2013, 191–201. ISBN: 9781450321594. DOI: 10.1145/2483760.2483785.

[121] M. Rocha, A. Simão, and T. Sousa. "Model-based test case generation from UML sequence diagrams using extended finite state machines". In: *Software Quality Journal* 29.3 (2021), pp. 597–627. DOI: 10.1007/s11219-020-09531-0.

[122] J. J. G. Rodriguez, M. J. E. Cuaresma, and M. M. Risoto. "A Model-Driven approach for functional test case generation". In: *Journal of Systems and Software* 109 (2015), pp. 214–228. DOI: 10.1016/j.jss.2015.08.001.

[123] K. Sakamoto, K. Shimojo, R. Takasawa, H. Washizaki, and Y. Fukazawa. "OCCF: A framework for developing test coverage measurement tools supporting multiple programming languages". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE. 2013, pp. 422–430. DOI: 10.1109/ICST.2013.59.

[124] D. Santiago, A. Cando, C. Mack, G. Nunez, T. Thomas, and T. M. King. "Towards Domain-Specific Testing Languages for Software-as-a-Service". In: *2nd International Workshop on Model-Driven Engineering for High Performance and Cloud computing (MDHPCL)*. 2013, pp. 43–52.

[125] D. C. Schmidt. "Guest Editor's Introduction: Model-Driven Engineering". In: *IEEE Computer* 39.2 (2006), pp. 25–31. DOI: 10.1109/MC.2006.58.

[126] E. Schoofs, M. Abdi, and S. Demeyer. "AmPyfier: Test Amplification in Python". In: *Journal of Software: Evolution and Process* (2022). DOI: 10.1002/smr.2490.

[127] L. Shan and H. Zhu. "Generating Structurally Complex Test Cases By Data Mutation: A Case Study Of Testing An Automated Modelling Tool". In: *The Computer Journal* 52.5 (2009), pp. 571–588. DOI: 10.1093/comjnl/bxm043.

[128] F. Siavashi, D. Truscan, and J. Vain. "Vulnerability Assessment of Web Services with Model-Based Mutation Testing". In: *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 2018, pp. 301–312. DOI: 10.1109/QRS.2018.00043.

[129] B. H. Smith and L. Williams. "On guiding the augmentation of an automated test suite via mutation analysis". In: *Empirical software engineering* 14.3 (2009), pp. 341–369. DOI: 10.1007/s10664-008-9083-7.

[130] M. Soden and H. Eichler. "Towards a Model Execution Framework for Eclipse". In: *Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture*. BM-MDA '09. New York, NY, USA: Association for Computing Machinery, 2009. ISBN: 9781605585031. DOI: 10.1145/1555852.1555856.

[131] H. A. de Souza, M. L. Chaim, and F. Kon. "Spectrum-based software fault localization: A survey of techniques, advances, and challenges". In: *arXiv preprint arXiv:1607.04347* (2016). DOI: 10.48550/arXiv.1607.04347.

[132] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[133] G. studio. *xFSM with ALE semantics*. 2021. URL: https://github.com/eclipse/gemoc-studio-execution-ale/tree/master/examples/language_workbench.

[134] G. studio. *xFSM with K3 semantics*. 2021. URL: https://github.com/eclipse/gemoc-studio/tree/master/official_samples/K3FSM/language_workbench.

[135] C.-a. Sun, Y. Liu, Z. Wang, and W. K. Chan. "μMT: A Data Mutation Directed Metamorphic Relation Acquisition Methodology". In: *Proceedings of the 1st International Workshop on Metamorphic Testing*. MET '16. Austin, Texas: Association for Computing Machinery, 2016, pp. 12–18. ISBN: 9781450341639. DOI: 10.1145/2896971.2896974.

[136] Tetrabox. *examples-behavioral-interface*. 2019. URL: https://github.com/tetrabox/examples-behavioral-interface/tree/master/languages/statemachines.

[137] Tetrabox-Gemoc. *MiniJava: a subset of Java as an executable DSL*. 2022. URL: https://github.com/gemoc/minijava.

[138] M. Tisi, J.-M. Mottu, D. S. Kolovos, J. De Lara, E. M. Guerra, D. Di Ruscio, A. Pierantonio, and M. Wimmer. "Lowcomote: Training the Next Generation of Experts in Scalable Low-Code Engineering Platforms". In: *STAF 2019 Co-Located Events Joint Proceedings: 1st Junior Researcher Community Event, 2nd International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems, and 1st Research Project Showcase Workshop co-located with Software Technologies: Applications and Foundations (STAF 2019)*. CEUR Workshop Proceedings (CEUR-WS.org). Eindhoven, Netherlands, July 2019. URL: https://hal.archives-ouvertes.fr/hal-02363416.

[139] P. Tonella. "Evolutionary Testing of Classes". In: *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '04. Boston, Massachusetts, USA: Association for Computing Machinery, 2004, pp. 119–128. ISBN: 1581138202. DOI: 10.1145/1007512.1007528.

[140] J. Troya, A. Bergmayr, L. Burgueño, and M. Wimmer. "Towards systematic mutations for and with ATL model transformations". In: *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2015, pp. 1–10. DOI: 10.1109/ICSTW.2015.7107455.

[141] J. Troya, S. Segura, and A. R. Cortés. "Automated inference of likely metamorphic relations for model transformations". In: *Journal of Systems and Software* 136 (2018), pp. 188–208. DOI: 10.1016/j.jss.2017.05.043.

[142] J. Troya, S. Segura, J. A. Parejo, and A. Ruiz-Cortés. "Spectrum-based fault localization in model transformations". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 27.3 (2018), pp. 1–50. DOI: 10.1145/3241744.

[143] M. Vieira, J. Leduc, W. M. Hasling, R. Subramanyan, and J. Kazmeier. "Automation of GUI Testing Using a Model-driven Approach". In: *Proceedings of the 2006 International Workshop on Automation of Software Test (AST)*. ACM, 2006, pp. 9–14. DOI: 10.1145/1138929.1138932.

[144] P. Vincent, K. Lijima, M. Driver, J. Wong, and Y. Natis. *Magic Quadrant for Enterprise Low-Code Application Platforms*. Tech. rep. 2019.

[145] T. Waheed, M. Z. Z. Iqbal, and Z. I. Malik. "Data Flow Analysis of UML Action Semantics for Executable Models". In: *Model Driven Architecture – Foundations and Applications*. Ed. by I. Schieferdecker and A. Hartman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 79–93. DOI: 10.1007/978-3-540-69100-6_6.

[146] K. Wang, A. Sullivan, D. Marinov, and S. Khurshid. "Fault localization for declarative models in Alloy". In: *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2020, pp. 391–402. DOI: 10.1109/ISSRE5003.2020.00044.

[147] S. Weißleder. "Simulated Satisfaction of Coverage Criteria on UML State Machines". In: *Third International Conference on Software Testing, Verification and Validation*. 2010, pp. 117–126. DOI: 10.1109/ICST.2010.28.

[148] W. E. Wong, V. Debroy, R. Gao, and Y. Li. "The DStar Method for Effective Software Fault Localization". In: *IEEE Transactions on Reliability* 63.1 (2014), pp. 290–308. DOI: 10.1109/TR.2013.2285319.

[149] W. E. Wong, V. Debroy, Y. Li, and R. Gao. "Software Fault Localization Using DStar (D*)". In: *2012 IEEE Sixth International Conference on Software Security and Reliability*. 2012, pp. 21–30. DOI: 10.1109/SERE.2012.12.

[150] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. "A survey on software fault localization". In: *IEEE Transactions on Software Engineering* 42.8 (2016), pp. 707–740. DOI: 10.1109/TSE.2016.2521368.

[151] A. Wortmann, O. Barais, B. Combemale, and M. Wimmer. "Modeling languages in Industry 4.0: an extended systematic mapping study". In: *Software and Systems Modeling* 19.1 (Sept. 2019), pp. 67–94. DOI: 10.1007/s10270-019-00757-6.

[152] H. Wu, J. Gray, and M. Mernik. "Unit Testing for Domain-Specific Languages". In: *Domain-Specific Languages*. Ed. by W. M. Taha. Springer Berlin Heidelberg, 2009, pp. 125–147. DOI: 10.1007/978-3-642-03034-5_7.

[153] T. Xie. "Augmenting Automatically Generated Unit-Test Suites with Regression Oracle Checking". In: *ECOOP 2006 - Object-Oriented Programming*. Ed. by D. Thomas. Vol. 4067. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 380–403. DOI: 10.1007/11785477_23.

[154] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu. "A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-Based Fault Localization". In: *ACM Transactions on Software Engineering and Methodology* 22.4 (2013). ISSN: 1049-331X. DOI: 10.1145/2522920.2522924.

[155] J. Xuan, X. Xie, and M. Monperrus. "Crash Reproduction via Test Case Mutation: Let Existing Test Cases Help". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, pp. 910–913. ISBN: 9781450336758. DOI: 10.1145/2786805.2803206.

[156] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. *The Fuzzing Book*. Retrieved 2021-10-26 15:30:20+02:00. CISPA Helmholtz Center for Information Security, 2021. URL: https://www.fuzzingbook.org/.

[157] H. Zhu. "JFuzz: A Tool for Automated Java Unit Testing Based on Data Mutation and Metamorphic Testing Methods". In: *2015 Second International Conference on Trustworthy Systems and Their Applications*. 2015, pp. 8–15. DOI: 10.1109/TSA.2015.13.

[158] P. Ziemann and M. Gogolla. "OCL Extended with Temporal Logic". In: *Perspectives of System Informatics*. Ed. by M. Broy and A. V. Zamulin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 351–357. ISBN: 978-3-540-39866-0. DOI: 10.1007/978-3-540-39866-0_35.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Un environnement de test pour les langages dédiés exécutables

Mots-clés: Ingénierie dirigée par les modèles (IDM), langage de modélisation dédié (LMD), tests de modèles, mesure de la qualité des tests, débogage des tests, amplification des tests

Résumé: La croissance continue de la complexité des logiciels soulève le besoin d'une gestion efficace de la complexité. L'ingénierie dirigée par les modèles (IDM) est un paradigme de développement qui répond à ce problème par la mise en place d'une séparation des préoccupations à l'aide de modèles. Un modèle est une abstraction spécifique d'un système qui peut être défini par un langage de modélisation dédié (LMD). Un LMD doté de fonctions d'exécution, appelé un LMD exécutable (LMDx), offre de de nouvelles possibilités dans l'activité de modélisation en permettant l'utilisation de techniques de vérification et de validation (V&V) dynamiques. Le test est la technique de V&V dynamique actuellement la plus répandue dans le domaine du génie logiciel. Bien qu'il existe de nombreux environnements de test pour les langages de programmation, produire un outillage de test pour un LMDx donné reste aujourd'hui une tâche coûteuse et difficile.

Dans cette thèse, nous proposons un environnement de test générique et réutilisable pour les LMD exécutables. Étant donné un LMDx, l'environnement fournit un langage de test qui prend en charge l'utilisation de concepts spécifiques au LMDx dans la définition de scénarios de test. Cela permet aux utilisateurs du LMDx, à savoir les experts du domaine, d'écrire des scénarios de test pour leurs modèles. Les scénarios de test écrits peuvent ensuite être exécutés sur les modèles, ce qui entraîne la production de résultats des tests. Pour aider davantage les experts du domaine à tester les modèles, l'environnement proposé offre trois services supplémentaires : (i) la mesure de la qualité des tests pour s'assurer que les scénarios de test écrits sont suffisamment bons ; (ii) le débogage des tests pour localiser le défaut du modèle testé en cas d'échec du test ; et (iii) l'amélioration automatique des tests pour renforcer la capacité des scénarios de test à détecter des régressions introduites dans les modèles testés.

A Testing Framework for Executable Domain-Specific Languages

Keywords: Model-Driven Engineering (MDE), Executable Domain Specific Language (xDSL), Model Testing, Test Quality Measurement, Test Debugging, Test Amplification

Abstract: The continuous growth of software complexity raises the need for effective complexity management. Model-Driven Engineering (MDE) is a development paradigm that meets this requirement by separating concerns through models. A model is a specific abstraction of a system that can be defined by a Domain-Specific Language (DSL). A DSL with execution facilities, referred to as Executable DSL (xDSL), enriches the modeling quality by enabling the employment of dynamic Verification & Validation (V&V) techniques. Testing is the most prevalent dynamic V&V technique in the field of software engineering. While many testing frameworks exist for general-purpose programming languages, providing testing facilities for any given xDSL remains a costly and challenging task.

In this thesis, we propose a generic testing framework for executable DSLs. Given an xDSL, the framework provides a testing language that supports the use of xDSL-specific concepts in the definition of test cases. This enables the xDSL's users, namely the domain experts, to write test cases for their models. The written test cases can be executed on the models and the test results will be produced. To further support the domain expert in *efficiently* testing models, the framework offers three supplementary services: (i) test quality measurement to ensure that the written test cases are good enough; (ii) test debugging to localize the fault of the model under test in case of test failure; and (iii) automatic test improvement to strengthen the ability of written test cases in detecting regression faults.