Frédéric Boulanger, Michalis Famelis and Daniel Ratiu, editors

Proceedings

11th Workshop on Model Driven Engineering, Verification and Validation MoDeVVa 2014

co-located with Models 2014

Valencia, Spain, September 30th 2014

Editor's addresses:

Frédéric Boulanger
Supélec - Département informatique
3 rue Joliot-Curie
91192 Gif-sur-Yvette cedex, France
frederic.boulanger@supelec.fr

Daniel Ratiu
Siemens AG
Corporate Technology
Otto-Hahn-Ring 6
81739 München, Deutschland
daniel.ratiu@siemens.com

Michalis Famelis
Department of Computer Science
10 King's College Road
University of Toronto
Toronto, Ontario, Canada M5S 3G4
famelis@cs.toronto.edu

Contents

Preface

The MoDeVVA workshop series brings together researchers and practitioners interested in combining MDE with validation and verification. The 11th edition took place on the 30th of September 2014 and was co-located with MODELS'14 in Valencia, Spain. The special topic of this edition was modeling and reasoning in the presence of incompleteness, underspecification and the unknown.

Out of the 19 papers submitted and reviewed by at least three members of the program committee, 9 were selected.

In addition to the presentation of the papers selected by the program committee, MoDeVVA'14 featured an invited presentation by Markus Völter, who works as an independent consultant in the domain of language engineering, modeling and model-driven software engineering.

This volume contains the abstract of the invited presentation and the final versions of the selected papers, in the order in which they were presented at the workshop. The papers where collected using the EasyChair conference system, formatted according to the LNCS style, and assembled using pdfLaTeX and the pdfpages package.

Program Committee

Ait Sadoune, Idir, *Supélec, France*
Balaban, Mira, *Ben-Gurion University of the Negev, Israel*
Barroca, Bruno, *CITI-FCT/UNL, Portugal*
Boulanger, Frédéric, *Supélec, France*
Bouquet, Fabrice, *University of Franche-Comté, France*
Bousse, Erwan, *Université de Rennes 1, France*
Cheng, Chih-Hong, *ABB Corporate Research, Germany*
Derrick, John, *University of Sheffield, UK*
Famelis, Michalis, *University of Toronto, Canada*
Fondement, Frederic, *École Nationale Supérieure d'Ingénieurs Sud Alsace, France*
Jacquet, Christophe, *Supélec, France*
Legeard, Bruno, *Smartesting, France*
Lúcio, Levi, *MSDL McGill University, Canada*
Merayo, Mercedes, *Universidad Complutense de Madrid, Spain*
Minea, Marius, *Universitatea Politehnica Timişoara, Romania*
Ratiu, Daniel, *Siemens Corporate Technology, Munich*
Salay, Rick, *University of Toronto, Canada*
Scheidgen, Markus, *Humboldt-Universitt zu Berlin, Germany*
Sokenou, Dehla, *GEBIT Solutions, Germany*
Taha, Safouan, *Supélec, France*
Williams, James, *University of York, Great Britain*
Wimmer, Manuel, *Vienna University of Technology, Austria*
Zurowska, Karolina, *Queen's University, Canada*

September 3, 2014
Gif-sur-Yvette

Frédéric Boulanger
Michalis Famelis
Daniel Ratiu

Language Workbenches: Opportunities and Challenges for V&V

Abstract of keynote speech at MoDeVVa 2014

Markus Völter

Voelter ingenieurbüro für softwaretechnologie
http://www.voelter.de/

Language workbenches (LWBs) are tools that support the efficient construction of languages. Several LWBs support modular extension and composition of languages, as well as flexibly mixing diverse notational styles in a single model. This has obvious advantages for V&V: languages of different levels of abstraction and levels of formality can be integrated, verification-specific extensions can be modularly introduced and validation is more efficient because programs can be much more readable as a consequence of domain-specific notations.

But there are also challenges:

- How can verification tools deal with (potentially unknown) extensions to the subject languages?
- Can the semantics implied by language extensions be exploited to speed up the verification?
- Can the semantics of extensions be proven to be equivalent to their lower-level representation?

In this talk I illustrate the opportunities an challenges based on mbeddr[1], an extensible set of integrated languages for embedded software development built with the JetBrains MPS language workbench.

My hope is that the opportunities help establish LWBs as a tool in the V&V community, and the challenges inspire discussions in the workshop and research after the conference.

[1] http://mbeddr.com/

Assessing the Quality of Meta-models

Jesús J. López-Fernández, Esther Guerra, and Juan de Lara

Universidad Autónoma de Madrid (Spain)
{Jesusj.Lopez, Esther.Guerra, Juan.deLara}@uam.es

Abstract. Meta-models play a pivotal role in Model-Driven Engineering (MDE), as they define the abstract syntax of domain-specific languages, and hence, the structure of models. However, while they play a crucial role for the success of MDE projects, the community still lacks tools to check meta-model quality criteria, like design errors or adherence to naming conventions and best practices.
In this paper, we present a language (*mmSpec*) and a tool (*metaBest*) to specify and check properties on meta-models and visualise the problematic elements. Then, we use them to evaluate over 295 meta-models of the ATL zoo by provisioning a library of 30 meta-model quality issues. Finally, from this evaluation, we draw recommendations for both MDE practitioners and meta-model tool builders.

1 Introduction

Model-Driven Engineering (MDE) considers models as the main assets in software development. Frequently, such models are not built using general-purpose modelling languages, like UML, but with Domain-Specific Languages (DSLs). Recent surveys on the use of MDE in industry [8] observe that nearly 40% of respondents use in-house DSLs, likely developed using meta-models. Hence, a critical factor in the success of MDE projects is the quality of the meta-models. However, the MDE community still lacks flexible tools permitting the specification, evaluation and user-friendly report of desired properties of meta-models.

A meta-model is considered of quality if it serves its purpose (contains all needed abstractions of the domain), and is technically built using sound principles (e.g., there are no repeated attributes among all sibling classes) [5]. The first concern is related to meta-model validation (*"are we building the right meta-model?"*) while the second refers to verification (*"are we building the meta-model right?"*).

In this paper, we present the language *mmSpec* and its supporting tool (*metaBest*), directed to the specification of desired meta-model properties, their evaluation, and the visualization of non-conforming parts of meta-models. We have used this tool to define a reusable library of 30 meta-model quality properties coming from quality criteria of conceptual schemas [2], naming guidelines [3], or from our experience. The properties have been classified in four categories depending on their relevance and nature. Moreover, we have evaluated the properties over 295 meta-models from the ATL zoo[1]. The ATL zoo is an open source

[1] http://www.emn.fr/z-info/atlanmod/index.php/Ecore.

repository to which any author can contribute, being the authorship of its meta-models attributed to a wide range of MDE community members. From the results of this study, we provide suggestions for MDE practitioners and meta-modelling tool builders.

This paper extends [9] (a tool demo paper) by presenting a library of quality properties that is evaluated over a repository of meta-models.

The remaining of this paper is organized as follows. First, Section 2 introduces the *mmSpec* language and tool, and overviews the library of quality properties. Section 3 presents the evaluation of the library over the meta-models of the ATL zoo. Section 4 compares with related research, and Section 5 ends with the conclusions and future work.

2 Specification of meta-model properties with *mmSpec*

We have developed a domain-specific language, named *mmSpec*, to specify meta-model properties and automate their evaluation on meta-models. In the following two subsections, we first introduce the main constructs of our language, and then we describe its use to construct a generic library of quality properties which can be applied on meta-models to assess their quality in an automatic way.

2.1 Definition and evaluation of meta-model properties

mmSpec allows expressing meta-model properties in a concise, intensional, declarative, platform-independent way. It provides high-level primitives that simplify the definition of meta-model properties, like first-order qualifiers for the length of navigation paths or collectors of the composed cardinality in navigation paths. Moreover, it is integrated with WordNet [11], which allows testing the nature of words (i.e., nouns and verbs) and synonymy.

The aim of this language and its companion tool (*metaBest* [9]) is providing a sound framework for expressing and evaluating quality issues and structural properties of meta-models. To favour simplicity, *mmSpec* properties follow a *select-filter-check* style that includes:

- A *selector* of the type (class, attribute, reference or path) and amount (a quantifier like every, some, none or an interval) of elements that should satisfy a given condition.
- An optional *filter* over the elements in the selector.
- A *condition* that is checked over the filtered elements.

Filters and conditions consist of *qualifiers* which can be negated, combined through and/or connectives, and point to new selectors, enabling recursive checks. The main qualifiers allow expressing conditions on the existence of elements, their name (nature, synonymy, prefix, suffix, camel-phrase), abstractness, multiplicity, type, length of navigation paths, inheritance relationships, depth and width of hierarchies and trees of containment relationships, collectors of the composed

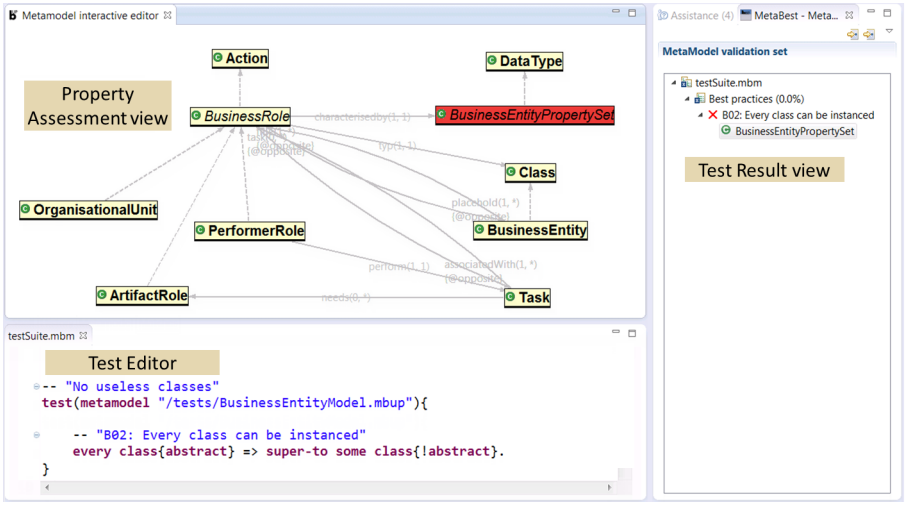


Fig. 1. Defining and evaluating a meta-model property.

cardinality in navigation paths, reachability from/to classes, and (a)cyclicity. Altogether, *mmSpec* promotes first-class primitives for elements (like paths or inheritance hierarchies) that need to be checked in meta-models frequently.

To evaluate a property on a meta-model, our Java-coded evaluator gathers the meta-model elements that match the property filter (or all of them, if there is no filter), verifies which ones meet the condition, and checks whether the size of the resulting subset is consistent with the selector's quantifier.

The bottom left of Fig. 1 shows a property using *mmSpec*. It states that every class (selector) that is abstract (filter) should have some concrete child class (condition), or as it is expressed, be *super to some class that is not abstract*. If an abstract class does not fulfil this condition, it is useless because it cannot be instantiated. The meta-model in Fig. 1 (taken from the zoo) does not satisfy the property, as class BusinessEntityPropertySet is abstract with no children.

Each property can be assigned a description of its intention, which gets shown in the *Test Result* view (see right of Fig. 1). While this view summarizes the results of all evaluated properties, the *Property Assessment* view highlights the relevant elements that did (green-coloured) or did not (red-coloured) fulfil a particular property. In this case, the class BusinessEntityPropertySet is displayed in red when we double-click on the property in the *Test Result* view.

The language offers abstraction mechanisms to package properties into functions with parameters. When the functions are called, it is possible to include a set of elements in place of a parameter. For example, fun(every class{!abstract}) evaluates property fun for all concrete classes in the meta-model.

Finally, *metaBest* supports batch evaluation of a set of properties over a set of meta-models, generating a CSV file with the results. In this way, we have managed to deliver the results of evaluating a library of meta-model quality issues for a large number of meta-models, as we explain in the following sections.

Code	Description
	Design
D01	An attribute is not repeated among all specific classes of a hierarchy.
D02	There are no isolated classes (i.e., not involved in any association or hierarchy).
D03	No abstract class is super to only one class (it nullifies the usefulness of the abstract class).
D04	There are no composition cycles.
D05	There are no irrelevant classes (i.e., abstract and subclass of a concrete class).
D06	No binary association is composite in both member ends.
D07	There are no overridden, inherited attributes.
D08	Every feature has a maximum multiplicity greater than 0.
D09	No class can be contained in two classes, when it is compulsorily in one of them.
D10	No class contains one of its superclasses, with cardinality 1 in the composition end (this is not finitely satisfiable).
	Best practices
BP01	There are no redundant generalization paths.
BP02	There are no uninstantiable classes (i.e., abstract without concrete children).
BP03	There is a root class that contains all others (best practice in EMF).
BP04	No class can be contained in two classes (weaker version of property D09).
BP05	A concrete top class with subclasses is not involved in any association (the class should be probably abstract).
BP06	Two classes do not refer to each other with non-opposite references (they are likely opposite).
	Naming conventions
N01	Attributes are not named after their feature class (e.g., an attribute paperID in class Paper).
N02	Attributes are not potential associations. If the attribute name is equal to a class, it is likely that what the designer intends to model is an association.
N03	Every binary association is named with a verb phrase.
N04	Every class is named in pascal-case, with a singular-head noun phrase.
N05	Element names are not too complex to process (i.e., too long).
N06	Every feature is named in camel-case.
N07	Every non-boolean attribute has a noun-phrase name.
N08	Every boolean attribute has a verb-phrase (e.g., isUnique).
N09	No class is named with a synonym to another class name.
	Metrics
M01	No class is overloaded with attributes (10-max by default).
M02	No class refers to too many others (5-max by default) – a.k.a. efferent couplings (Ce).
M03	No class is referred from too many others (5-max by default) – a.k.a. afferent couplings (Ca).
M04	No hierarchy is too deep (5-level max by default) – a.k.a. depth of inheritance tree (DIT).
M05	No class has too many direct children (10-max by default) - a.k.a. number of children (NOC).

Table 1. Library of meta-model quality properties.

2.2 A library of quality properties for meta-models

In order to test the quality of meta-models, we have built an *mmSpec* library covering typical mistakes (some of them from [2]) that designers tend to commit, as well as others that may jeopardize a basic level of meta-model quality. The library has four categories of issues, depending on their nature and relevance:

Design. Properties signalling a faulty design (an error).
Best practices. Basic design quality guidelines (a warning).
Naming conventions. For example, use of verbs, nouns or pascal/camel case.
Metrics. Measurements of meta-model elements and their threshold value, like the maximum number of attributes a class should reasonably define. Most metrics are adapted from the area of object-oriented design [6].

Table 1 lists the properties from these categories. To illustrate *mmSpec*'s expressiveness, next we show the formulation of a property from each category:

– *D02: There are no isolated classes.* The encoding of this property is:

no class => **and** { **sub**−**to no class**, **super**−**to no class**,
reach no class, **reached**−**from no class** }.

The aim is to check the absence of classes that are not involved in any association or hierarchy. Thus, we use the no class selector, and check the following conditions: the class is orphan (qualifier sub-to with selector no class), childless (qualifier super-to with selector no class), contains no reference (qualifier reach with selector no class), and is not pointed by any other (qualifier reached-from with selector no class).

– *BP03: There is a root class that contains all others.* This is a common best-practice in EMF, where meta-models define a class from which all other classes can be reached through composition relations. Its encoding is:

strictly 1 **class** {**cont**−**root** {**absolute**}} => **exists**.

A class satisfying cont-root is the root of a containment tree; if the root is absolute, then it contains all classes. This illustrates how *mmSpec* provides primitives that simplify the definition and checking of meta-model properties.

– *N04: Every class is named in pascal-case, with a singular-head noun phrase.* To obtain intuitive class names, these should be composed by a sequence of words starting with capital letters, and with a singular noun as the last word [3]. For instance, WashingMachine is a good class name, but Washing_Machine and MachineWashing are not. The connection of *mmSpec* with WordNet enables checking whether a word is a singular noun.

every class => **name** = **pascal**−**phrase**{**end**{**noun**{**singular**}}}.

– *M01: No class is overloaded with attributes.* Even in large meta-models, classes with too many attributes often evidence a questionable design. While some entities in certain domains might carry a vast load of information, commonly, this data can be split into smaller entities that are arranged using inheritance or composition. Thus, the following property states that every class should have a maximum of 10 non-inherited (!inh) attributes.

every class => **with** {**!inh**} [0, 10] **attribute**.

3 Assessing the quality of existing meta-models

To evaluate our tool and have a measure of the quality of current meta-modelling practice, we have applied our library of quality properties to the Atlan Ecore zoo of 295 meta-models. We have chosen this repository as it is representative of the meta-models that MDE practitioners build in practice. The size of the meta-models varies from tiny ones with just one class, to meta-models of medium size, the largest one having 699 classes. This is interesting as one of our goals is to detect whether the kind of quality issues depends on the meta-model size, and whether big meta-models are faultier (even if in average) than smaller ones.

Fig. 2 shows the number of quality issues detected in the analysed meta-models. Interestingly, only 5 meta-models have no issue, while no meta-model contains more than 22. The average number of issues per meta-model is 7.26.

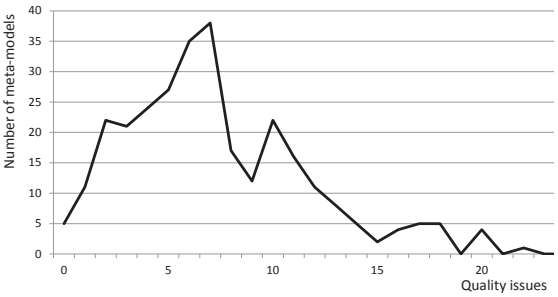


Fig. 2. Number of quality issues in meta-models.

Regarding the distribution of issues according to their kind, Fig. 3 shows how many meta-models fail each property from Table 1. *Design* is the most relevant category of properties, as it gathers errors that may potentially lead to a faulty design. In this sense, the results for the properties in this category are good in average, as they have low rate of failure. Indeed, there are two *design* properties that every meta-model fulfils: D01 and D02. D01 checks the absence of repeated attributes in a hierarchy (see *D01* in Fig. 4 for a faulty example), while D02 checks that the upper bound of features is not 0. However, 110 meta-models fail property D09 (37% of analysed meta-models). This error consists in making a class to be contained in two other classes, with minimum source multiplicity 1 in one of the containment relationships, as shown in Fig. 4. This is an error because, at the instance level, an instance of A could never be contained in an instance of C, as it must be mandatorily contained in an instance of B.

Surprisingly for a set of EMF meta-models, the top unmet property is BP03, an EMF best practice that states the need for a root class whose instances may contain the whole model tree. Fig. 4 shows an example meta-model that fulfils this property, and an example that does not. In *BP03 (+)*, A contains B and C, and hence D (as it is subclass of B), so A acts as absolute root class. On the contrary, *BP03 (-)* does not meet the property because A does not contain D.

The next two properties not satisfied by more meta-models are N03 and N04, which are naming conventions. N03 demands the verbalization of binary association names (e.g., reaches). N04 checks the conventions for class names, as explained in Section 2.2.

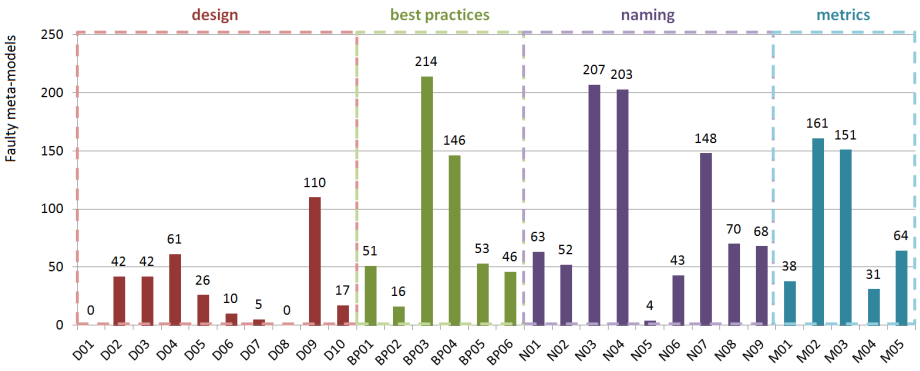


Fig. 3. Number of meta-models that contain issues of a certain type.

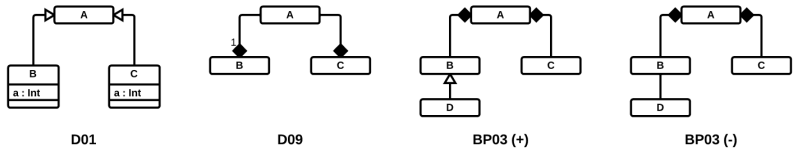


Fig. 4. Some quality issues of the library.

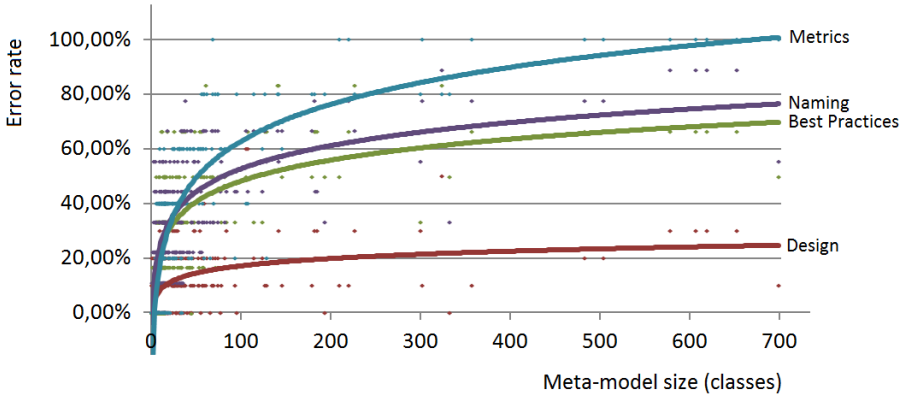
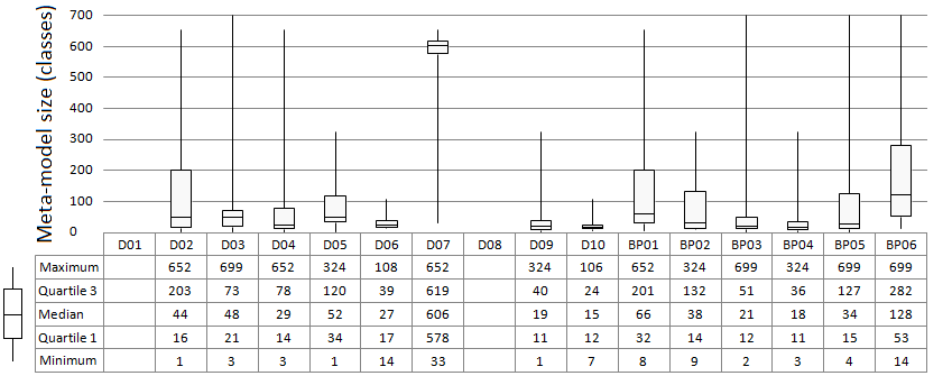


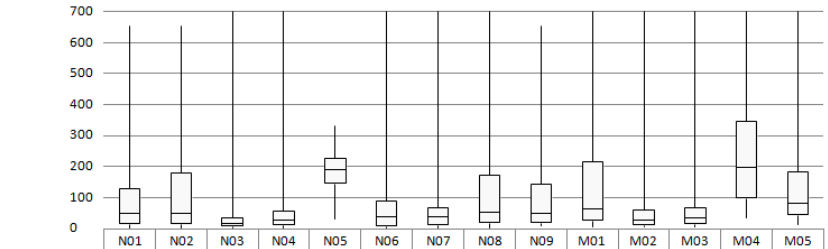
Fig. 5. Percentage of non-fulfilled issues in each category, w.r.t. meta-model size.

Regarding the error rate evolution of each category of issues with respect to the meta-model size (measured in number of classes), Fig. 5 shows that all categories present an upward error trend as meta-models enlarge. The vertical axis in this diagram corresponds to the percentage of issues in the category that were not met by some meta-model. The growth ratio is higher in small/medium size meta-models (up to 100 classes). Then, the error growth is steadier, particularly on the *design* category, which remains around 20% even at the largest meta-model size. The issues of type *metrics* grow as the meta-model size increases, peaking 100% (i.e., all issues of the category fail in large meta-models). This might be comprehensible, since a greater number of classes usually demands a greater number of features and relationships. However, properties such as M01, M02 or M03 might be considered independent from the meta-model size, as they take care of the class feature overpopulation, which is a bad practice despite the meta-model size. More worrisome is the evolution of the *best practice* category. This category reflects less severe concerns than *design*, but they still are bad modelling practices. It is worth noting that most meta-models present a 40 to 65% error rate, which together with the *design*'s - almost permanent - 20%, denote an average design quality that may be improved.

Fig. 6 shows the distribution of meta-model sizes where each issue type tends to occur. For most properties, faulty meta-models have between 1 and 200 classes. However, properties M04, BP06, D07 tend to fail in large meta-models. This is natural in some cases, e.g., D07 checks the presence of overridden at-



	D01	D02	D03	D04	D05	D06	D07	D08	D09	D10	BP01	BP02	BP03	BP04	BP05	BP06
Maximum		652	699	652	324	108	652		324	106	652	324	699	324	699	699
Quartile 3		203	73	78	120	39	619		40	24	201	132	51	36	127	282
Median		44	48	29	52	27	606		19	15	66	38	21	18	34	128
Quartile 1		16	21	14	34	17	578		11	12	32	14	12	11	15	53
Minimum		1	3	3	1	14	33		1	7	8	9	2	3	4	14



	N01	N02	N03	N04	N05	N06	N07	N08	N09	M01	M02	M03	M04	M05
Maximum	652	652	699	699	332	699	699	699	652	699	699	699	699	699
Quartile 3	128	180	35	57	228	90	68	173	143	217	61	70	345	185
Median	48	45	16	26	189	35	32	55	48	61	29	34	193	70
Quartile 1	16	18	10	14	147	10	13	20	21	28	14	16	101	47
Minimum	3	3	2	3	31	3	1	4	10	5	6	6	36	14

Fig. 6. Meta-model size dispersion by property.

tributes in inheritance hierarchies. Instead, properties D06, D09, D10, BP04, N03 tend to occur in small meta-models. The fact that 3 important design properties fail in small meta-models might mean that such meta-models were built by more unexperienced designers, compared to large ones. As a matter of fact, properties with a greater number of failure occurrences (as seen in Fig. 3: D09, BP03, BP04, N03, N04, N07, M02, M03) mainly appear in really small meta-models.

Finally, if we look at the average number of quality issues per class, we find that the most frequent categories of issues are *best practices* and *naming conventions* (0.18 issue occurrences per class in both cases). *Design* (0.07 issues per class) and *metrics* (0.05 issues per class) are less frequent; nonetheless, if they are considered together, the error rate seems worrisome at least.

3.1 Discussion

From the analysis of the meta-model repository, we realize that a way to improve the quality of meta-models is the inclusion of these quality checks in the meta-modelling tools, for example, to discover problems like D09. Actually, for some of these problems (like the ones related to metrics) the tool could trigger some refactoring suggestions. Regarding naming conventions, we noticed the usefulness

of having integrated "smart" spell checkers (i.e., to check correctness of names in camel-case).

It is worth mentioning that *mmSpec* is integrated with *metaBup* [10], a tool for the example-based construction of meta-models. This means that the meta-model builder can check these quality issues during meta-model construction.

4 Related work

We could use OCL instead of *mmSpec*. However, OCL expressions tend to be more complex, as OCL lacks primitives (for gathering paths or inheritance hierarchies) which are part of *mmSpec* and have been designed to express properties on meta-models. Moreover, *mmSpec* supports the visualization of problematic elements (i.e., properties do not report just true/false). A comparison of OCL and *mmSpec* is available at: `http://www.miso.es/tools/metaBest.html`.

In [7], quality properties are defined as QVT-Relations transformations which produce a model with the problems in a meta-model. They define a catalog of problems for MOF-based meta-models, categorized into: syntactic (i.e., well-formedness constraints), semantic (i.e., poor design choices), and convention. Interestingly, *mmSpec* fulfils the features that [7] demands from any automated model verification approach: it is declarative, generic, flexible (though not standard), direct, and it has easy-to-inspect reporting facilities. Some of its primitives are specific for meta-model verification and would be difficult to specify with QVT-Relation patterns.

In [2, 4], quality properties of conceptual schemas are formalised in terms of quality issues, which are conditions that should not happen. Our approach also aims at detecting errors or bad smells; however, we focus on meta-models (not schemas). Thus, our library considers all issues in [2] that are meaningful in meta-modelling, as well as others specific to meta-models (like the existence of a root for EMF meta-models). While in [2], the method is evaluated on schemas developed by students, our library is applied to a public repository of meta-models built by developers. The same authors propose guidelines for naming UML schemas in [3], for which they provide a tool [1]. Interestingly, [3] presents a study on the effectiveness of current UML modelling environments for building schemas (not meta-models), and concludes that by including more quality issues in the IDEs, the quality of the developed schemas increases.

Some works aim at characterizing meta-model quality. For example, in [5], the authors adapt the ISO/IEC 9126 for meta-models, proposing concepts like completeness, conciseness, detailedness or complexity. However, there is no concrete proposal on how to measure such properties.

Few works analyse the quality of real meta-models. In [13], the authors take some basic size metrics (e.g., number of classes) over meta-models from different repositories (including the ATL zoo). In the same line, [12] correlates meta-model metrics, like the usage of inheritance w.r.t. meta-model size. Instead, we focus on detecting patterns that may indicate flaws in meta-models.

5 Conclusions and future work

In this paper, we have introduced *mmSpec*, a language directed to the specification of properties to be checked on meta-models, and *metaBest*, a tool to visualize and report the problematic elements. We have used both to build a catalog with 30 meta-model quality properties, which has been evaluated over 295 meta-models. The obtained results show that most meta-models contain some issue; hence, the community would benefit from integrated tool support (like *metaBest*) for checking quality properties during meta-model construction.

In the future, we plan to analyse correlations between meta-model flaws, provide a catalog of quick fixes and recommendations, and support the creation of user-defined categories for properties. We also plan to develop a further language for meta-model testing based on constraint solving.

Acknowledgements. This work has been funded by the Spanish Ministry of Economy and Competitivity with project "Go Lite" (TIN2011-24139).

References

1. D. Aguilera, R. García-Ranea, C. Gómez, and A. Olivé. An eclipse plugin for validating names in UML conceptual schemas. In *ER Workshops*, volume 6999 of *LNCS*, pages 323–327. Springer, 2011.
2. D. Aguilera, C. Gómez, and A. Olivé. A method for the definition and treatment of conceptual schema quality issues. In *ER'12*, volume 7532 of *LNCS*, pages 501–514. Springer, 2012. See also `http://helios.lsi.upc.edu/phd/catalog/issues.php`.
3. D. Aguilera, C. Gómez, and A. Olivé. A complete set of guidelines for naming UML conceptual schema elements. *Data Knowl. Eng.*, 88:60–74, 2013.
4. D. Aguilera, C. Gómez, and A. Olivé. Enforcement of conceptual schema quality issues in current integrated development environments. In *CAiSE*, volume 7908 of *LNCS*, pages 626–640. Springer, 2013.
5. M. F. Bertoa and A. Vallecillo. Quality attributes for software metamodels. In *QAOOSE'10*, 2010.
6. S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994.
7. M. Elaasar, L. C. Briand, and Y. Labiche. Domain-specific model verification with QVT. In *ECMFA*, volume 6698 of *LNCS*, pages 282–298. Springer, 2011.
8. J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of MDE in industry. In *ICSE*, pages 471–480. ACM, 2011.
9. J. J. López-Fernández, E. Guerra, and J. de Lara. Meta-model validation and verification with MetaBest. In *ASE*, pages 1–4 (to appear). ACM, 2014.
10. J. J. López-Fernández, J. Sánchez Cuadrado, E. Guerra, and J. de Lara. Example-driven meta-model development. *SoSyM*, in press, 2014, see also `http://www.miso.es/tools/metaBUP.html`.
11. G. A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, 1995.
12. J. D. Rocco, D. D. Ruscio, L. Iovino, and A. Pierantonio. Mining metrics for understanding metamodel characteristics. In *MiSE*, pages 55–60. ACM, 2014.
13. J. R. Williams, A. Zolotas, N. D. Matragkas, L. M. Rose, D. S. Kolovos, R. F. Paige, and F. A. C. Polack. What do metamodels really look like? In *EESS-MOD@MoDELS*, volume 1078 of *CEUR*, pages 55–60, 2013.

Towards Scalable Multidimensional Execution Traces for xDSMLs

Erwan Bousse[1], Benoit Combemale[2], and Benoit Baudry[2]

[1] University of Rennes 1, France
erwan.bousse@irisa.fr
[2] Inria, France,
{benoit.combemale, benoit.baudry}@inria.fr

Abstract. Executable Domain Specific Modeling Languages (xDSML) opens many possibilities in terms of early verification and validation (V&V) of systems, including the use of *dynamic* V&V approaches. Such approaches rely on the notion of *execution trace*, *i.e.* the evolution of a system during a run. To benefit from dynamic V&V approaches, it is therefore necessary to characterize what is the structure of the executions traces of a given xDSML. Our goal is to provide an approach to design trace metamodels for xDSMLs. We identify seven problems that must be considered when modeling execution traces, including concurrency, modularity, and scalability. Then we present our envisioned approach to design scalable multidimensional trace metamodels for xDSMLs. Our work in progress relies on the *dimensions* of a trace (*i.e.* subsets of mutable elements of the traced model) to provide an original structure that faces the identified problems, along with a trace API to manipulate them.

1 Introduction

In the recent years, a lot of effort have been made to provide tools and methods to design executable Domain Specific Languages (xDSMLs) [4,5]. Executability of models opens many possibilities in terms of early verification and validation (V&V) of systems, including the possiblity to rely on *dynamic* V&V approaches such as debugging, runtime verification [6] or model checking [2].

A central concept in dynamic V&V approaches is the *execution trace*, which represents the evolution of a system during a run. A trace is the alternate sequence of *states* of the system and *events* that triggered the state changes. All previously cited approaches rely on traces: model checking consists in verifying a property of a system by analyzing all its possible traces, and a counter-example in the form of a trace is provided if it is not satisfied; runtime verification consists in checking whether or not a trace satisfies a property; debuggers require traces to be able to replay faulty scenarios in order to investigate for bugs.

Henceforth, a significant prerequisite for the V&V of executation models is the definition of the structure of the execution traces of a considered xDSML. Since an xDSML should define what is the state (also called *runtime data*) of

a model during its execution [4], a *trace metamodel* could potentially be defined based on this information. But many questions remain regarding what data should contain a trace, or how it should be structured. In this paper, we identify a number of problems to face when modeling execution traces, and we present an approach to face these problems. We first introduce in Section 2 a motivating example, which is an xDSML called RoboML and a simple scenario involving two robots following one another. Then in Section 3 we list and illustrate a series of seven problems linked to trace modeling, including concurrency, modularity and scalability. In Section 4 we present our approach to generate scalable and multidimensional trace metamodels. Finally Section 5 concludes with perspectives.

2 Motivating example

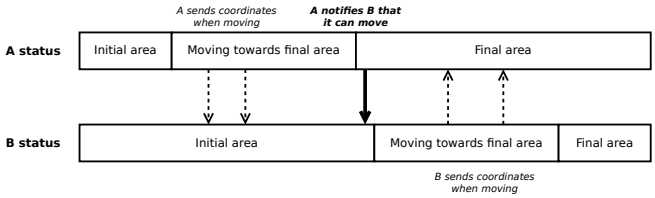


Fig. 1. Scenario of two robots following one another

Our motivating example consists in an xDSML called RoboML, which extends timed automata with hierarchy, events, and domain specific actions. A RoboML model consists in a set of communicating timed automata. When executed, the current state of the automata changes depending on the current state, events, clocks and conditions. Additional domain-specific runtime data is available, such as the GPS coordinates of the robot. Actions triggered on transitions allow a robot to move, interact and communicate with other robots. The very simple scenario we consider is based on two robots A and B, each configured with a RoboML model. Robots are configured to regularly send their coordinates to the other robot when they change their position. Figure 1 illustrates the scenario, which can be summed up in the following way:

1. Both robots are in an initial area.
2. Robot A moves towards a final area.
3. When arrived, A waits 5 seconds and then sends a message to B to move.
4. Then B moves to the final area.

We want to check the following two properties:

(a) B starts moving 5 seconds after A, not before nor after.
(b) Each time a robot covers 1 meter, it sends a message to the other one with its new coordinates.

3 Problems when Modeling Traces

In this section, we present a series of problem that must be dealt when modeling execution traces, and we illustrate them with the example introduced in the previous section.

3.1 Trace Contents

The first category of problems concerns the contents of an execution trace, apart from the state of the executed model and the events that triggered state changes.

(Pb. 1) Concurrency modeling The execution of one or multiple models may imply concurrent variables within runtime data. In such case, the different states of these variables may or may not be independent from one another. In our example, the status of a robot is most of the time completely independent from the status of the other. However, communications between the robots imply some kind of ordering between the states of the robots. For instance, we know that B didn't start moving before receiving a message from A, thus allowing us theoretically to verify property (a). A challenge is thus to take into account these concurrency relationships between variables into the trace structure.

(Pb. 2) User-defined additional information Analyzing the behavior of a system requires information about how it changes over time. Yet, in some cases, properties may concern data that we do not directly have. In our example, property (b) concerns the meters covered by a robot, which is not directly available in the state of the executed model. Such variable could easily be derived from the evolution of the coordinates of the robot and would belong in the trace to verify such properties.

(Pb. 3) Scalability in space Execution traces can be arbitrarily large, as some complex systems are monitored continuously in case a failure occurs. Thus a challenge is to manage scalability in space when manipulating traces, whether it is offline (file or database storage) or online (in memory). In our example, a robot changes its internal state all the time to update its coordinates or listen to communications, leading quickly to a large amount of states.

3.2 Trace Manipulations

The second category of problems concerns the eventual manipulations of an execution trace, which may constrain the structure of the trace.

(Pb. 4) Modularity A trace must be *constructed* during the run of a program, in order to be *manipulated* either during or after the run. Modularity of the trace is a problem for both the construction and the manipulation phases. First, when tracing a system, one can be interested in observing only a subset of the variables, which require a non-monolithic and modular trace format. Second,

when manipulating a trace, one might want to extract a subset of the information (e.g. a subtrace with only a selection of variables), or conversely to add new information within the trace (e.g. derived variables). In our example, property (a) only concerns the movement states of the robots and not their coordinates, thus extracting a trace with only the former information would be relevant to prove this property.

(Pb. 5) Manipulation safety Generic trace metamodels exist to model all kinds of traces for any executable language [1]. However, constructing traces with such metamodels may lead to inconsistent trace models, since their genericity does not forbid one to create a trace whose states are not relevant to the concerned xDSML. In our example, for instance, we may want to ensure that coordinates are stored as a pair if integers instead of a string.

(Pb. 6) Reuse of trace manipulations To verify properties on traces, or to be able to write interesting queries to explore them, we must define operations that manipulate traces. An important need is to be able to reuse such operations from a trace to another, from a system to another, or even from an xDSML to another. In our case, verifying property (a) requires an operator that checks all states that are found 5 seconds after a specific situation, which can be generalized in a *within* operator.

(Pb. 7) Scalability in time Traces are eventually analyzed, which requires iterating over the steps of the trace. The potentially large size of a trace compromises the capacity to make queries in a reasonable time. Moreover, if some variable referenced in a property only changes lately in a trace, we would still have to iterate through all steps before noticing that change. This is the case with property (a) of our example, where B starts moving very lately.

4 Envisioned Approach

In this section, we present the multiple choices we made for our approach to design scalable multidimensional trace metamodels, and how these choices participate in solving the problems stated in the previous section.

4.1 Trace Structure

Our approaches relies on an original way to structure execution traces. We illustrate it informally in Figure 2, and we highlight important choices in the following paragraphs.

Multiple dimensions A trace can be defined as a single alternate sequence of *states* and *events*. Yet, it is very likely that only a subset of the variables really change from a state to another. Our idea is thus to consider multiple *dimensions* in a trace, each being a set of mutable elements of the executed model. We then

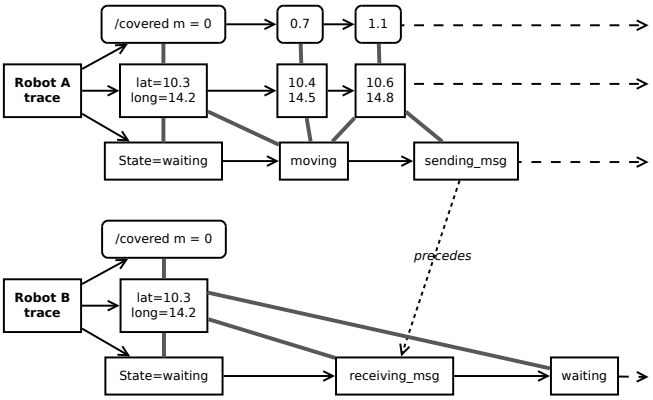


Fig. 2. Intuitive and partial representation of a multidimensional trace, matching the very beginning of the robots scenario (until the first coordinates message is sent).

define a trace as a set of *subtraces*, each being the evolution of a specific dimension within the run. Such structure allow us to solve many problems. First, by manipulating dimensions separately, we can define concurrency relationships between them (Pb. 1). More precisely, we consider that states can be linked by *observations* (*i.e.* states that were simultaneous at some point) or *synchronizations* (*i.e.* states that changed simultaneously), while events can be linked by ordering relationship such as *precedence* or *coincidence* (we refer to [7] for more relationships between events). Second, we gain modularity (Pb. 4), as we can add or remove dimensions depending on the needs. Third, such modularity helps enriching the trace with additional dimensions (Pb. 2). Finally, this structure allow us to iterate separately through different dimensions, which should improve scalability in time (Pb. 7). Figure 2 shows an example: our robot trace consists in six dimensions (three per robot: covered meters, coordinates, state). We have a precedence relationship between the sending and the receiving of the message. Covered meters being not in the runtime data, it is added as a new dimension derived from coordinates.

Data sharing For scalability in space (Pb. 3), our idea is to maximize data sharing among steps of the trace, *i.e.* reduce redundancy from a step to another. Our approach relies on ideas of our previous work on scalable model cloning [3]. More precisely, each dimension state is stored in a dedicated storage structure in order to be referenced by the steps of the same dimension.

Domain specific trace metamodel To provide manipulation safety to trace models (Pb. 5), our solution is to design *domain specific* trace metamodels, *i.e.* metamodels each of which defines precisely what are the traces of a single xDSML. This choice ensures that all traces are consistent with regard to the traced language. In addition, we can define the state of a model of a given

xDSML without relying on unsafe generic types (e.g. *EObject* when using the Eclipse Modeling Framework (EMF)). The drawback is that is becomes necessary to provide one trace metamodel per xDSML, but we plan to overcome this by providing a generative approach.

4.2 Trace API

Generating trace metamodels has multiple advantages as stated in the previous section. However, the main drawback is that operations defined for a given trace metamodel are not compatible with a different trace metamodel (Pb. 6). Following the same trend as recent first-class traces approaches [8], our solution is to generate, along with a trace metamodel, a complete API with trace specific operations in order to refine, query, explore, or transform a trace. Operations such as *filter*, *merge*, *slice*, *during*, etc. are part of this API.

5 Conclusion

Verification and validation of executable models is a challenge that requires the modeling of execution traces. We identified seven problems that must be considered when modeling traces, and we presented our idea of multidimensional traces coupled with a trace manipulation API. Such traces take into account concurrency, scalability, modularity among other aspects. Further work will be the implementation of the approach and its application to RoboML.

Acknowledgement. This work is partially supported by the ANR INS Project GEMOC (ANR-12-INSE-0011).

References

1. Luay Alawneh and A Hamou-Lhadj. Execution traces: A new domain that requires the creation of a standard metamodel. *Advances in Software Engineering*, 2009.
2. Christel Baier and Joost-Pieter Katoen. *Principles Of Model Checking*.
3. Erwan Bousse, Benoit Combemale, and Benoit Baudry. Scalable Armies of Model Clones through Data Sharing. *MODELS 2014, Valencia, Spain*, 2014.
4. Benoît Combemale, Xavier Crégut, and Marc Pantel. A Design Pattern to Build Executable DSMLs and associated V&V tools. *The 19th Asia-Pacific Software Engineering Conference*, 2012.
5. Benoit Combemale, Julien Deantoni, Matias Vara Larsen, Frédéric Mallet, Olivier Barais, Benoit Baudry, and Robert France. Reifying Concurrency for Executable Metamodeling. In *International Conference on Software Language Engineering*, 2013.
6. Martin Leucker and Christian Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 2009.
7. Frédéric Mallet. Clock constraint specification language: Specifying clock constraints with UML/MARTE. *Innovations in Systems and Software Engineering*, 4, 2008.
8. Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Expositor: Scriptable time-travel debugging with first-class traces. *ICSE 2013, San Francisco, CA*, 2013.

Verifying Modelling Languages using Lightning: a Case Study

Loïc Gammaitoni, Pierre Kelsen, and Fabien Mathey

University of Luxembourg

Abstract. The formal language Alloy was developed to provide fully automatic analysis of software designs. By providing immediate feedback to users it allows early detection of design errors. The main goal of the Lightning tool is to apply the power of Alloy's automatic analysis to the domain of software language engineering. The tool allows to represent abstract syntax, concrete syntax and semantics of a modelling language in Alloy. In this paper we describe the verification capabilities of Lightning with the help of a concrete modelling language, namely the language of structured business processes.

1 Introduction

The formal language Alloy was developed to "capture the essence of software abstractions simply and succinctly, with an analysis that is fully automatic, and can expose the subtlest of flaws" [9]. By allowing continuous automatic analysis during the design process software modellers can uncover design errors quickly. This design process, which could aptly be called "agile modelling", also stimulates the modellers since it provides immediate feedback.

The goal of the Lightning tool[1] is to apply the power of the Alloy language and its tool, the Alloy Analyzer, to the domain of software language engineering. It was already shown earlier [12] that Alloy is a suitable language for defining syntax and semantics of modelling languages . The Lightning tool can be viewed as a first practical validation of ideas presented in that work.

One can consider Lightning as an important step towards a language workbench based on Alloy. The emphasis of the tool is currently on automatic validation of language definitions using Alloy's SAT-based analysis. All basic components of a modelling language can be defined in the tool: abstract syntax, concrete syntax and semantics. Concrete syntax is currently restricted to visualising language models. Semantics can be specified in the style of operational semantics and its execution can be visualised as well. All specifications of language components and accompanying transformations are defined in Alloy. The tool is, however, not limited to language specifications expressed in Alloy since it allows importing metamodels expressed in Ecore (feature not described in the present paper). For Lightning to become a full-fledged language workbench [5], more sophisticated editor support has to be provided (only exists in rudimentary form in the present version) as well as code generation facilities to interface with existing programming languages.

The main purpose of this paper is a description of the verification capabilities of the Lightning tool. We will examine how the tool assists the user in writing correct language specifications.

This paper is organised as follows: we first describe the case study we will use in this paper. In section 3 we introduce the Lightning tool. We then describe how Lightning assists the user in designing the abstract syntax (section 4), concrete syntax (section 5), and semantics (section 6). We wrap up the paper with a discussion of our contribution in the context of related work and present concluding remarks and future work in the final section.

2 Case Study

In this paper, we illustrate Lightning's verification features by designing a Structured Business Process (SBP) language.

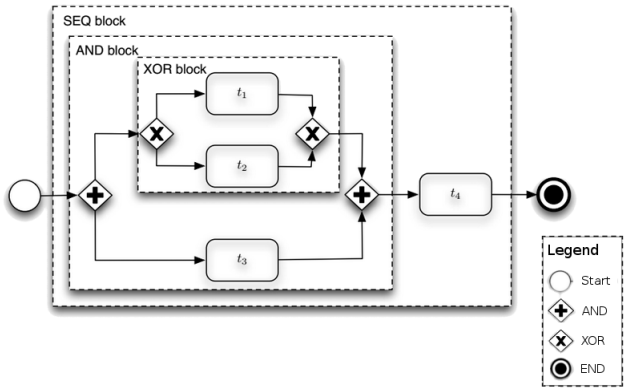


Fig. 1. A Structured Business Process

SBPs consist of *tasks* representing actions performed towards the completion of the process and of *control nodes* structuring the process. Those tasks and control nodes are interconnected using transitions so that the following holds:

- The process has a unique start and end, represented by the Start and End control nodes, so that no transition is incoming to Start or outgoing from End.
- Each task has exactly one incoming and one outgoing transition.
- XOR and AND are control nodes used to delimit blocks representing the nesting of processes. The difference between XOR and AND is purely semantical. While AND means that all sub-processes (outgoing transitions) need to be processed, XOR specifies that exactly one of them has to be processed.

- XOR and AND control nodes have one incoming and more than one outgoing transition if they are used to open a new block (in which case they are called XOR split and AND split), or more than one incoming and one outgoing transition if they are used to close a new block (in which case they are called XOR join and AND join)
- A Block opened by an AND split or XOR split needs to be closed by an AND join or XOR join, respectively.
- The process is acyclic (all tasks are traversed at most once)

An example business process representing a model expressed in this language is represented in fig. 1 using traditional notation from the business process community.

This choice of case study is based on the fact that:

- The SBP's specification has been formalized in [17], thus providing a precise description of the syntax and semantics of this language.
- It has sufficient complexity to illustrate the usefulness of our tool.
- It is practically relevant since many existing business processes are expressible in this form [17].

This case study has been implemented using Lightning in the context of a master thesis. The concrete verification examples presented in this paper have actually been encountered during that work.

3 Lightning

The Lightning tool is a language workbench based on Alloy. It is distributed as an Eclipse plugin [1]. It provides support to formally express all the components of a language (Abstract Syntax, Concrete Syntax, and Semantics), and allows to verify these using Alloy's SAT based model finding mechanism. Amongst the notable features of Lightning are :

- A complete Alloy editor (with outline, error markers and syntax highlighting)
- Ecore support
- An editor allowing to modify generated instances.

The signature trait of Lightning, however, is to allow incremental language development (depicted in fig. 2) by coupling the instance generation of Alloy with the domain specific visualization and model execution induced by the concrete syntax and semantics definition, respectively . This approach facilitates the identification of design errors [7].

In the following sections, we will delve into the details of the design process shown in fig. 2 and describe associated verification tasks.

[1] Freely available at : http://lightning.gforge.uni.lu

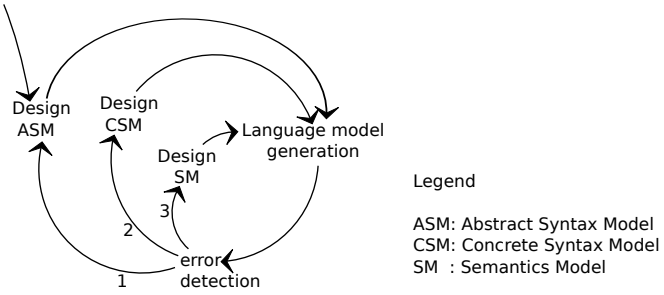


Fig. 2. Spiral diagram depicting how languages are incrementally designed in Lightning

4 Abstract Syntax Design

In Lightning the abstract syntax of a language consists of an Alloy model defining the set of valid language models. We can view this model as the *metamodel* of the language. In our SBP case study, the abstract syntax model (ASM) defines concepts of the language (Tasks, Flows, Control nodes, ...), relations between those concepts (e.g., Flows have Nodes as source and target), and well-formedness rules expressed as constraints (e.g., to ensure acyclicity of the process). The following is an excerpt of the abstract syntax model:

```
1  abstract sig Node{}
2  abstract sig Control extends Node{}
3  one sig Start extends Node{}{this not in Flow.target}
4  one sig End extends Node{}{this not in Flow.source}
5  sig Task extends Node{}
6  sig AND_JOIN, AND_SPLIT, XOR_JOIN, XOR_SPLIT extends Control{}
7  sig Flow{
8    source: Node,
9    target: Node
10 }
11 fact acyclic{
12   all n: Node | n not in n.^((~target).source)
13 }
```

Listing 1.1. Abstract Syntax Model excerpt

We can use Alloy's instance generation mechanism to verify the abstract syntax. This scenario corresponds to the cycle labelled **1** in fig. 2. Figure 3 depicts one of the language models thus obtained from our SBP specification. Although it is still possible to interpret this model correctly, it is a bit tedious since it is not presented in the traditional way but reflects the structure of the abstract syntax. The more complex a language model is (in terms of number of elements and links present), the harder it becomes to comprehend it. This is why it is advised to start defining the concrete syntax of a language (transit to cycle 2 in fig. 2) once its models become hard to check through their default visualization.

In the next section we define how domain specific visualizations are specified in Lightning.

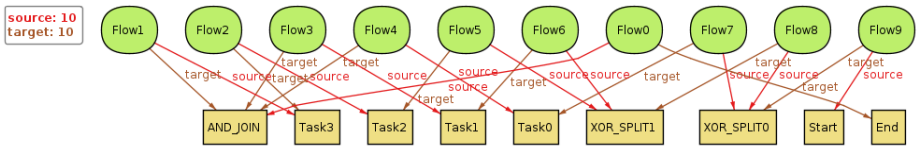


Fig. 3. Raw visualization of a language model (using Alloy's Magic Layout)

5 Concrete Syntax Design

The Concrete Syntax of a language consists of an Alloy model defining a transformation from the previously defined Abstract Syntax Model (ASM) to a predefined Visual Language Model (VLM). This definition follows the approach that Kleppe describes in [13]. This VLM, named LightningVLM and also expressed in Alloy, consists of:

- A set of visual elements that can be linked and composed
- Layout and color declarations that can be used as properties of visual elements
- Well-formedness rules that enforce that any instance can be correctly rendered once interpreted by the tool (by preventing the presence of cyclic compositions, for example)

The transformation model enforces that all of its instances contain a given ASM instance and its corresponding VLM instance via the use of mapping rules and integration predicates; these predicates specify the values of fields of atoms in the VLM instance. The VLM instance can then be interpreted by Lightning in order to be rendered graphically. This process is the essence of the concrete syntax support the tool provides and is depicted in fig. 4. Note that in the current version of Lightning the concrete syntax is used only for visualisation and cannot be directly edited.

In order to be processed in a reasonable time, the Alloy model defining this ASM to VLM transformation can be written following a sub-syntax of Alloy, such that interpretation can be used rather than SAT-solving. This approach called functional module is introduced in [8].

Below we provide a selection of the mapping rules and their integration predicates (prefixed with the prop_ keyword) defined in order to provide a concrete syntax to our SBP language.

```
1  /* each task is represented by a rectangle, and each node has its
       corresponding label */
2  one sig Transformation{
3      mapTask: Task one -> one RECTANGLE,
4      mapNodeText: Node one -> one TEXT
5  }
6  /* a task is represented by a rectangle with a white
7  background that contains the corresponding text */
8  pred prop_mapTask(n: Task, r: RECTANGLE) {
9      r.layout = VERTICAL_LAYOUT
10     r.color = WHITE
```

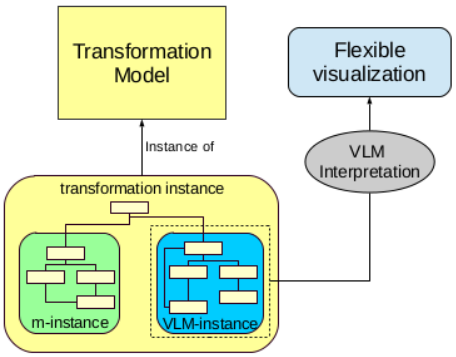


Fig. 4. Visualization process using a transformation from ASM to VLM

```
11      r.composedOf[0] = Bridge.mapNodeText[n]
12  }
13  /* the text is black, not styled, and is labeled
14  after the label of the node it represents */
15  pred prop_mapNodeText(n:Node, t:TEXT) {
16      t.color = BLACK
17      t.isItalic = False
18      t.isBold = False
19      t.textLabel[0] = n
20  }
```

Once the concrete syntax is defined in this way, it becomes easier to detect errors in an instance model. Figure 5 depicts the language model previously shown in fig. 3, visualized this time using its concrete syntax definition.

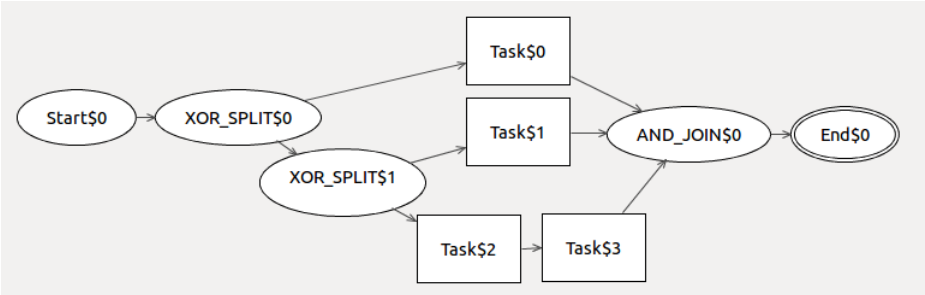


Fig. 5. Visualization of the instance depicted in fig. 3 using its concrete syntax definition

Only one glance at fig. 5 suffices to notice that our SBP language is underspecified. Indeed, in this language model, two XOR splits are converging into

a single join. Moreover this join, which is an AND join, doesn't have the same nature than the converging splits. In order to fix this design error, we need to associate splits and joins together. We do this via the definition of control boxes:

```
sig ControlBox {
    split: Control,
    join: Control
}{ (
    // SPLIT AND JOIN HAVE SAME NATURE
    split in AND_SPLIT and join in AND_JOIN) or
   (split in XOR_SPLIT and join in XOR_JOIN)
    // PAIRING EACH SPLIT WITH A GIVEN JOIN
    all s: (succ[split]) | s in (preds[join])
    all j: (pre[join]) | j in (succs[split])
}
```

Adding the concept of a control box to the abstract syntax and repeating the instance generation shows us that the error has been well identified and fixed. The error processing we just discussed illustrates a transit to the cycle 1 of fig. 2., i.e., to the case where an error found in the visualisation reveals an error in the underlying abstract syntax. Of course the transformation model describing the visualisation may be faulty itself. In this case the error in the visual representation may point to an error in the concrete syntax model. This situation corresponds to a transit to the cycle 2 of fig. 2, leading to redesigning the Concrete Syntax model. Checking if the error seen in the concrete syntax visualization is also present in the concrete-syntax-less visualization (described in the previous section) allows to decide whether or not the error has been introduced by the concrete syntax definition.

6 Semantics Definition

Lightning currently offers the possibility to define the operational semantics of languages.

The semantics definition in Lightning consists of:

- a Semantics Model (SM) in which the concepts of *state* and *trace* are defined. A step predicate is specified that expresses the condition that one state follows another state in the trace.
- a Semantics visualization transformation model, reusing most of the rules present in the ASM to VLM transformation but adding rules to represent the properties of the semantics state.

For our case study each state consists of a set of nodes that are currently active in the execution of the business process. The corresponding field of the Alloy signature is called *currentNodes*. That is, for a given state s, the expression *s.currentNodes* denotes the set of active nodes in state s. The visualisation represents the currently active nodes by highlighting them in the business process model.

To verify the correctness of the operational semantics, one can visualize its possible executions. To illustrate this verification, let us consider the following predicate as a first attempt to define the semantics of XORs:

```
1 pred XORNodes(current:Node,s2:State) {
2     current in XOR_SPLIT and one node: current.(~source).target | node
          in s2.currentNodes
3 }
```

This predicate ensures that given a current node that is a XOR_SPLIT, the set of current nodes belonging to the next semantics state contains exactly one of the nodes directly following the XOR_SPLIT (mutual exclusion). Figure 6 gives an example of an erroneous execution.

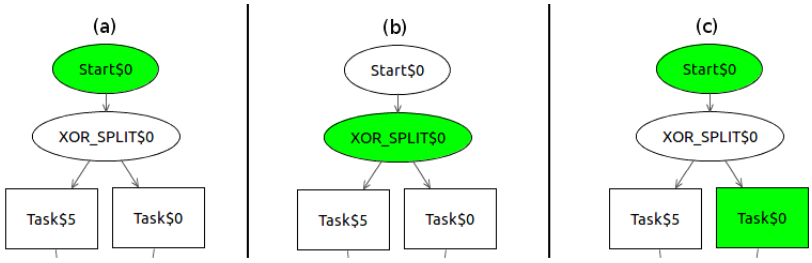


Fig. 6. Erroneous execution of a business process model

Although the transition from (a) to (b) is performed as expected, the transition from (b) to (c) shows us that our XOR semantics is underspecified. Indeed, the predicate previously shown enforces that only one of the nodes directly following an active XOR_SPLIT should be part of the current nodes. This predicate thus does not specify the state of the other nodes, thus allowing extraneous nodes to appear in the set of current nodes for a given state. To fix this, one simply needs to enforce that the set of current nodes of a given state is contained in the set of successors of all the current nodes present in the previous semantics state (code omitted for lack of space).

The example above illustrates the case where an error in the concrete syntax representation of the semantic state points to an error in the underlying semantic model. This case corresponds to the cycle 3 of fig. 2.

7 Discussion and Related Work

The term "language workbench" was made popular by Martin Fowler [6]; it denotes a tool that supports the efficient definition, reuse and composition of languages and their IDEs [5]. The Lightning tool may be viewed as a language workbench that is based on the formal language Alloy (although not a full-fledged one as mentioned in the introduction). Because of its formal basis it differs from existing language workbenches such as MetaEdit+[11], MPS[18], and Spoofax[10]. Few workbenches currently support formal semantic analysis; notable exceptions are Kermeta [2] and Atom 3 [4] for which some formal analysis

is available via a translational semantics (to Maude for Kermeta [3] and to Alloy for AToM3 [2], [19]).

Our work is based on the premise that developing modelling languages benefits from the lightweight formal modelling approach offered by Alloy because it gives language developers immediate feedback on design decisions using automatic formal analysis and thus allows to detect design errors early. We can thus view Lightning as an attempt to provide agile modelling of software languages in a way similar to the initial intent of Alloy, namely providing agile modelling of software designs.

Because our tool is based on Alloy it also inherits the inherent limitations of Alloy. Indeed verification is based on instance finding via SAT solving. The effectiveness of this approach intimately depends on the small scope hypothesis, stating that most of the design errors can be found in small models. Assuming the small scope hypothesis holds, the approach will allow to reduce the scopes of signatures in Alloy so that a correct answer can be found in reasonable time. Of course a negative answer in the search of a counterexample does not exclude the possibility that there may be one but may point instead to the need for trying out larger scopes, resulting of course in longer running times.

In the context of language design, though every aspect of a language is written in Alloy, the performance limitations of Alloy we just mentioned only apply to the generation of language models (ASM instances). The visualisation and semantics, benefiting from functional modules, can be processed efficiently [8].

8 Conclusion

We have presented in this paper how Lightning allows the application of a lightweight verification technique based on Alloy from the earliest stages of a domain specific language design process to its completion. In particular we have given concrete examples of verification tasks that were carried out during the design of a language for structured business processes.

Regarding future work much remains to be done. One obvious hindrance to the use of our tool is the fact that it requires prior knowledge of Alloy. We are currently trying to see to what extent we can provide graphical interfaces to most of the modelling tasks in the tool. In particular we have already partially implemented such an interface for defining transformations.

Another fundamental question that needs to be investigated concerns performance. Indeed, once the metamodel becomes a bit larger (with, say, tens of signatures) Alloy's instance generation tends to slow down appreciably. Recent work on model slicing (such as [14, 15]) in the context of UML/OCL models) suggests that in many cases instance generation can be made more efficient by generating instances for subparts of the metamodel and then combining these partial instances into an instance of the whole metamodel. We plan to investigate this type of approach in the context of our work.

[2] A newer version of the tool named AToMPM [16] is available

References

1. Lightning tool web site, http://lightning.gforge.uni.lu.
2. Kermeta tool web site, http://www.kermeta.org.
3. Moussa Amrani. A formal semantics of kermeta. *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, 2012.
4. Juan De Lara and Hans Vangheluwe. Atom3: A tool for multi-formalism and meta-modelling. In *Fundamental approaches to software engineering*, pages 174–188. Springer, 2002.
5. Sebastian et al. Erdweg. The state of the art in language workbenches. In Martin Erwig, Richard F. Paige, and Eric Wyk, editors, *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 197–217. Springer International Publishing, 2013.
6. Martin Fowler. Language workbenches: The killer-app for domain specific languages. http://martinfowler.com/articles/languageWorkbench.html.
7. Loïc Gammaitoni and Pierre Kelsen. Domain-specific visualization of alloy instances. In *ABZ*, pages 324–327, 2014.
8. Loïc Gammaitoni and Pierre Kelsen. Functional Alloy Modules. Technical Report TR-LASSY-14-02, University of Luxembourg; http://hdl.handle.net/10993/16386.
9. Daniel Jackson. *Software abstractions*. MIT Press Cambridge, 2012.
10. Lennart CL Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and IDEs. In *ACM Sigplan Notices*, volume 45, pages 444–463. ACM, 2010.
11. Steven Kelly, Kalle Lyytinen, and Matti Rossi. Metaedit+ a fully configurable multi-user and multi-tool CASE and CAME environment. In *Advanced Information Systems Engineering*, pages 1–21. Springer, 1996.
12. Pierre Kelsen and Qin Ma. A lightweight approach for defining the formal semantics of a modeling language. In *Model Driven Engineering Languages and Systems*, pages 690–704. Springer, 2008.
13. Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 2008.
14. Asadullah Shaikh, Robert Clarisó, Uffe Kock Wiil, and Nasrullah Memon. Verification-driven slicing of uml/ocl models. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 185–194. ACM, 2010.
15. Asadullah Shaikh, Uffe Kock Wiil, and Nasrullah Memon. Uost: Uml/ocl aggressive slicing technique for efficient verification of models. In *System Analysis and Modeling: About Models*, pages 173–192. Springer, 2011.
16. Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Hüseyin Ergin. Atompm: A web-based modeling environment. In *Demos/Posters/StudentResearch@ MoDELS*, pages 21–25, 2013.
17. Silvano Colombo Tosatto, Guido Governatori, and Pierre Kelsen. Towards an abstract framework for compliance. *Proceedings of the 17th IEEE International EDOC 2013 Conference Workshops*, pages 79–88, 2013.
18. Markus Voelter and Vaclav Pech. Language modularity with the mps language workbench. In *34th International Conference on Software Engineering (ICSE)*, pages 1449–1450. IEEE, 2012.
19. Thomas De Vylder. Feature modelling: A survey, a formalism and a transformation for analysis. University of Antwerp.

Scalable Verification of Model Transformations

Xiaoliang Wang, Adrian Rutle, and Yngve Lamo

Bergen University College (Norway)
{xwa, aru, yla}@hib.no

Abstract Model transformations are crucial in model driven engineering (MDE). Automatic execution of model transformations improves software development productivity. However, model transformations should be verified to ensure that the models produced or the transformations satisfy some expected properties. In a previous work we presented a verification approach of graph-based model transformation systems based on relational logic. The approach encodes model transformation systems as Alloy specifications which are examined by the Alloy Analyzer. But experiments showed scalability and performance problems in the approach when complex relations were present in the systems. To solve these problems, we extend our previous work by using three techniques: 1) we change the encoding to decrease the arity of relations in the derived Alloy specifications; 2) we *decompose* the expressions for the pattern matching into sub-expressions using *unique elements*; 3) we use *annotations* to decrease the complexity of the metamodel and the model transformation rules. The results of our experiments indicate that the new techniques lead to better scalability and performance.

Keywords: Model transformations; verification of model transformations; scalability problem; verification performance.

1 Introduction

In model driven engineering (MDE), models are the first class entities of the software development. They are used to specify the domain under study, to generate program code, to document software, etc. Ideally, models in a development phase can be generated automatically from models in a previous phase by model transformations. Such automation makes MDE appealing by increasing productivity. However, errors may exist in model transformations and be propagated to subsequent phases resulting in erroneous models and software. Thus, verification of model transformations is a necessary task to ensure correctness, i.e. the produced models or the transformations satisfy some expected properties.

In our previous work [22], we presented a verification approach of graph-based model transformation systems based on relational logic. A model transformation system, which consists of a metamodel and model transformation rules, is encoded automatically as an Alloy specification. Then the specification is examined by the Alloy Analyzer to verify if the system satisfies some properties within a user-defined scope. We verified conformance, safety and reachability properties

by checking the *Direct Condition* and the *Sequential Condition* [22]. However, the previous approach suffered from a scalability problem. Complex relations in metamodels and transformation rules caused relations of high arity present in the derived Alloy specifications. As a consequence, the Alloy Analyzer could not examine the specifications when using larger scopes.

To handle the scalability problem we change the encoding procedure, especially the part for matching the left and right patterns of the model transformation rules, to get rid of relations with high arity. A side effect of this change is that we get longer expressions for the pattern matching. This leads to increased verification time and worse performance. To handle the performance problem we extend our approach with two techniques: decomposition and annotation. With the first technique, we divide the expressions for the pattern matching into sub-expressions using *unique elements*. With the second technique, we use annotations to specify state information. This will decrease the complexity of the relations and the amount of the constraints in the metamodel.

Section 2 recalls the verification approach and presents some related background information. Sections 3, 4 and 5 present the changes in the encoding procedure, the usage of unique elements and the usage of annotations, respectively. In Section 6, we describe several experiments and present the results to show the effect of the new techniques. Related work is discussed in Section 7 and finally concluding remarks and future work are presented in Section 8.

2 Background

In this section, we recall the verification approach for graph-based model transformation systems detailed in [22]. Since the verification approach is based on Alloy [1] and the running example is specified in the Diagram Predicate Framework (DPF) [11,12,19,21], we give first a brief introduction to these frameworks.

2.1 Alloy

Alloy is a specification language and an analyzing tool for relational models. The language is declarative, suited for describing complex model structures and constraints based on relational logic. Artifacts in a model are defined in Alloy as *signatures* while relations among them are defined as fields of the signatures. Some predefined signatures are offered, like *Int*. Constraints on specifications are defined as *fact*s while reusable constraints with parameters are defined as *pred*s. The Alloy Analyzer is a constraint solver which verifies Alloy specifications by first translating them into SAT problems and then resolving them using a SAT solver. It can find instances which are well-typed by (and are satisfying the constraints of) the specifications using the *run* command. One can also use the *check* command to find counterexamples violating some properties. Notice that Alloy performs a bounded check, i.e. for each signature, a user-defined *scope* bounds the number of its instances.

2.2 Diagram Predicate Framework (DPF)

DPF provides formalization of (meta)modelling and model transformation based on graph transformations [13] and category theory [5]. (Meta)models in DPF are

represented by diagrammatic specifications consisting of an underlying graph and a set of constraints. Fig. 1 is the metamodel of Dijkstra mutual exclusion algorithm [10], in which P, T, R and $\{F1, F2\}$ represent process, turn, resource and the flags of a process while `[xor]` and `[inj]`, etc. are the constraints [22] Each constraint is formulated by a predicate from a predefined signature. A model is valid, i.e. the model conforms to its metamodel if there is a typing morphism from the underlying graph of the model to the underlying graph of the metamodel, and the model satisfies all the constraints of the metamodel.

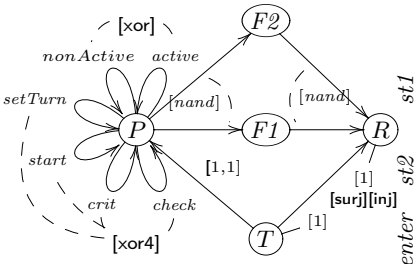


Figure 1: Metamodel

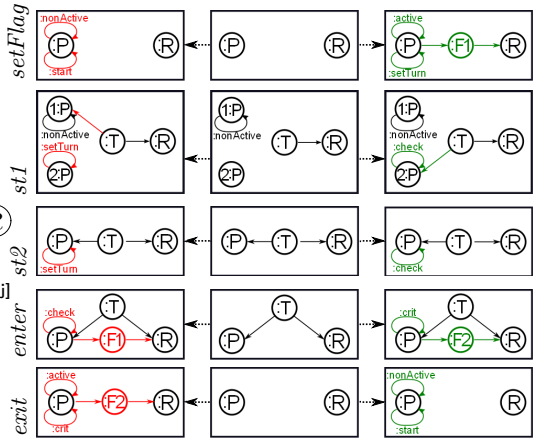


Figure 2: Transformation rules

A model transformation system consists of a metamodel (Fig. 1) together with a set of model transformation rules (Fig. 2). The transformation rules specify how a source model can be transformed to a target model. A model transformation rule $p : L \xleftarrow{l} K \xrightarrow{r} R$, consists of the left pattern L, the right pattern R and the coordination pattern K, together with two injective graph morphisms l and r. A model transformation consists of a sequence of direct model transformations (application of one model transformation rule). The execution of transformations follows the classic Double-Pushout (DPO) approach as in [13]. That is, for each match m of L in the source model S, we create a match n of R in T using two pushout constructions. DPF also provides a framework to specify constraint-aware model transformation rules [21]. That is, the transformation rules may contain constraints which may be used to control the matches and to decide what to create in the target model.

2.3 Verification Approach

In [22], as a running example we represented Dijkstra's mutual exclusion algorithm as a model transformation system (see Fig. 1 and 2). The approach uses an encoding procedure to translate the system to an Alloy specification. The metamodel and the transformation rules are encoded as corresponding signatures and predicates in the Alloy specification as follows:

1. Assume that there are m nodes and n edges in the structure of the metamodel. Each node N_i, $i \in [1..m]$ is encoded as a signature S_{N_i} while each edge E_j, $j \in [1..n]$ as a signature S_{E_j} with two fields $src : one\ E_j^s$ and $trg : one\ E_j^t$. The Alloy keyword *one* encodes that each edge has exactly one source (target) node E_j^s (E_j^t). Models are encoded as signatures S_G with two fields $nodes : set\ S_{N_1} + \cdots + S_{N_m}$ and $edges : set\ S_{E_1} + \cdots + S_{E_n}$ encoding the contained nodes and edges.
2. The DPF predicates are encoded as *pred*s in Alloy while constraints as $fact$s.
3. Direct model transformations are encoded as a signature *Trans* with 7 fields: the rule applied *rule*, the source and target model *source*, *target*, as well as, the deleted and added elements : *dns* and *ans* (nodes), *des* and *aes* (edges).
4. Each rule application is encoded as a predicate $rule_i[trans]$ stating that the transformation *trans* applies rule i. In this predicate, we encode that the source (target) model has exactly one match of the left (right) pattern in which some matched elements are deleted (added) according to the rule.
5. In order to ensure that each direct model transformation applies only one rule, for each type in the metamodel, a number restricts how many of its instances are deleted (added).

```
1 sig S_{N_i}{} //For each node N_i, i∈[1..m]
2 sig S_{E_j}{src:one E_j^s, trg:one E_j^t} //For each edge E_j, j∈[1..n]
3 sig S_G{nodes:set S_{N_1} + ... + S_{N_m}, edges:set S_{E_1}+... + S_{E_n}}
4 sig Trans{rule:one Rule, source,target:one Graph, dns,ans:set S_{N_1}+...+S_{N_m},
      des, aes:set S_{E_1}+...+S_{E_n}}
5 fact{all t:Trans | rule_1[t] or ... or rule_r[t]}
```

After applying the encoding procedure, we verify the system by using the Alloy Analyzer to examine the specification. The Alloy Analyzer searches for counterexamples by executing the command $check\{constraint\}\ for\ scope$. If no counterexample is found, the property is verified correct within the scope. Otherwise, a counterexample can be visualized to assist the designer to correct the system. In [22] we verified conformance, safety and reachability properties by checking the *Direct Condition* (i.e. every direct model transformation produces a valid target model) and the *Sequential Condition* (i.e. if the direct condition is not satisfied, for each counterexample there exists a sequence of direct model transformations that will produce a valid target model).

2.4 Challenges in Verification

The Alloy Analyzer performs a bounded check within a state space determined by the *scope*. Given an Alloy specification consisting of m signatures, a scope $[s_1, s_2, ..., s_m]$ bounds the size of the ith signature to s_i. For a relation of arity n, the size of the state space containing all the possible instances is $2\hat{}(\sum_{i=1}^{m} s_i)^n$ [16]. The encoding procedure, especially the encoding of the pattern matching, leads to relations of high arity in the Alloy specifications. This impairs scalability since relations of high arity hampers verification within large scopes, e.g. in [22] the highest scope was 4. In the following section, we present a new encoding for the pattern matching which reduces the arity of the relations in the Alloy specifications.

3 Changes in the Encoding Procedure

A direct model transformation is the application of one model transformation rule. As a result, the source (target) model has a match of the left (right) pattern of the rule with elements deleted (added) according to the rule. Assuming that the left (right) pattern of a transformation rule is a connected graph containing edges $e_1, \ldots, e_m$ and nodes $v_1, \ldots, v_n$ (if the pattern is a disconnected graph, each connected subgraph is encoded separately), the pattern matching is encoded as $one\ e_1, \ldots, e_m | p(e_1, \ldots, e_m)$, where $p(e_1, \ldots, e_m)$ is the relational expression about structure match and change. The Alloy keyword *one* enforces that only one pattern is present in the source (target) model. For example, the right pattern of the rule *setFlag* in Table 2 is encoded as follows:

```
one a:active&t.aes, s:setTurn&t.aes, pf:PF1&t.aes, f:F1R&t.aes |
 let p=a.src, f1=pf.trg, r=f.trg |
 p=a.trg and s.src=p and s.trg=p and pf.src=p and f.src=f1 and f1 in F1&t.ans
  and r in R&(t.source.nodes-t.dns) and p in P&(t.source.nodes-t.dns)
```

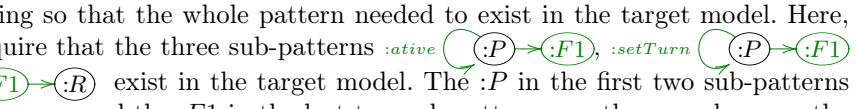
The quantification $one\ e_1, \ldots, e_m$ in the relational formula causes relations of high arity in the Alloy specification. The Alloy Analyzer cannot handle such quantification with larger scopes, especially when using the keyword *one* [22]. After studying the encoding procedure, we find that the keyword *one* is not necessary because the number restriction on the deleted (added) elements implies that only one pattern is present in the source (target) model (see item 5 in Section 2.3). Therefore, we could use *some* instead of *one*. Since the expression $some\ e_1, \ldots, e_m | p(e_1, \ldots, e_m)$ equals to $some\ e_1 | \ldots | some\ e_m | p(e_1, \ldots, e_m)$, we can split the existential quantification and evaluate the quantifiers one by one. In this way, we obtain Alloy specifications without high-arity relations. For example, the right pattern of the rule *setFlag* can now be encoded as:

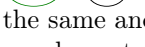
```
some a:active&t.aes|let p=a.src|
 p in P&(t.source.nodes-trans.dns) and p=a.trg
  and some s:setTurn&t.aes|s.src=p and s.trg=p
  and some pf:PF1&t.aes|let f1=pf.trg|pf.src=p and f1 in F1&t.ans
  and some f:F1R&t.aes|let r=f.trg|f.src=f1 and r in R&(t.source.nodes-t.dns)
```

Thus, the scalability problem due to high-arity relations can be solved by the above mentioned change of the encoding. However, this leads to longer expressions for the pattern matching, which in turn will lead to poor performance during verification. In the following two sections, we introduce two techniques to address this problem and improve the performance. Note that in Section 6 we summarize the gains in translation time and verification time due to the optimization techniques discussed in the following sections.

4 Decomposition of Patterns

In order to get rid of long expressions for the pattern matching, we decompose them into sub-expressions during the encoding procedure using unique elements. A deleted (added) element is unique if it is the only instance of its type deleted (added) during a direct model transformation. Usually some unique elements

share a common structure. Consider the right pattern of the rule *setFlag*, in a transformation applying the rule, the added arrow :*active* in Fig. 2 is the only instance of *active* which will be added. The edges :*setTurn* and :*P*→:*F1* are unique elements sharing the same :*P*. In addition, the unique elements :*P*→:*F1* and :*F1*→:*R* share the same :*F1*. In the previous section, we encoded the pattern matching so that the whole pattern needed to exist in the target model. Here, we require that the three sub-patterns 

and 

exist in the target model. The :*P* in the first two sub-patterns are the same and the :*F*1 in the last two sub-patterns are the same because the unique elements share them. After decomposition using the unique elements, the right pattern of the rule *setFlag* is encoded as:

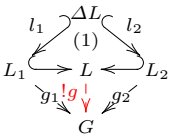
```
some a:active&t.aes|let p=a.src|p in (t.source.nodes-t.dns) and p=a.trg
 and some pf:PF1&t.aes|let f1=pf.trg|pf.src=p and f1 in t.ans
some s:setTurn&t.aes|let p=s.src|p in (t.source.nodes-t.dns) and p=s.trg
 and some pf:PF1&t.aes|let f1=PF10.trg|pf.src=p and f1 in t.ans
some e:F1R&t.aes|let f1=e.src,r=e.trg|f1 in t.ans and r in (t.source.nodes-t.
    dns)
```

Now we use the following lemma to prove that the decomposition is valid.

Lemma 1. *Assume a pattern L is divided into two sub-patterns L_1 and L_2 where $L_1 \cup L_2 = L$ and $L_1 \cap L_2 = \Delta L$ containing unique elements. In addition, the source and target nodes of the edges contained by ΔL are also contained by ΔL. We can prove that G has a match of L iff G has matches of g_1:L_1→G and g_2:L_2→G, where $g_1|_{\Delta L} = g_2|_{\Delta L}$.*

Proof. ⇒ Straightforward

⇐ l_1:ΔL→L_1, l_2:ΔL→L_2 are the two inclusion maps. Due to the assumptions, the square (1) in the right figure is a pushout. Since $g_1|_{\Delta L} = g_2|_{\Delta L}$, we can get $l_1; g_1 = l_2; g_2$. Therefore, there is a unique map $!g$:L→G.



In the new encoding procedure we identify sub-patterns sharing the same unique elements. In this way, long expressions for the pattern matching are decomposed into sub-expressions, thus gaining better translation time.

5 Annotation in Transformation Systems

In this section, we use annotations to specify state information in the metamodel and the transformation rules. The encoding procedure is changed correspondingly. The complexity of the metamodel is decreased since we reduced the amount of signatures and constraints.

5.1 Changes of Metamodel and Rules

In Section 2, arrows (e.g. *nonActive*, *start*) and nodes (e.g. *F0*) were used to specify the states of a process. In this way, complex relations appeared in the models. Furthermore, we needed to add extra constraints to correctly specify states; the metamodel in Figure 1 had 28 constraints. In order to reduce the number of the signatures and constraints, we use annotations to specify the

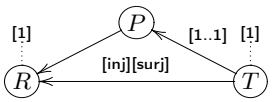


Figure 3: Metamodel

Predicate	Visualization
<Flag>	:P <F0>|<F1>|<F2>
<IsActive>	:P <active>|<nonActive>
<At>	:P <start>|<setTurn>|<check>|<crit>

Table 1: The signature Σ_2 used for annotation

Rule	L	K	R
setFlag	<nonActive> <F0> :P :R <start>	:P :R	<active> <F1> :P :R <setTurn>
st1	<nonActive> 1:P :T :R 2:P <setTurn>	<nonActive> 1:P :T :R 2:P	<nonActive> 1:P :T :R 2:P <check>
st2	:P :T :R <setTurn>	:P :T :R	:P :T :R <check>
enter	:T <check> :P :R <F1>	:T :P :R	:T <crit> :P :R <F2>
exit	<active> <F2> :P :R <crit>	:P :R	<nonActive> <F0> :P :R <start>

Table 2: Transformation Rules

state information; the new metamodel contains only 10 constraints. A similar technique was used in [20] to specify states of workflow instances.

Fig. 3 shows the metamodel using the annotation technique. The nodes R, P, T are the same as in Fig. 1. Annotations, e.g. <nonActive>, <Active> and <F1>, are used to specify state information instead of graph structures (like nodes and edges). We use three groups of annotations; for instance the group <Flag> with three annotations <F0>, <F1> and <F2>. Each group has an applicable type t in the metamodel. In an instance, each element typed by t can be marked with only one annotation from the group. In the example, all the annotation groups can be applied to instances of the type P.

Since the transformation rules in a model transformation system are defined based on the metamodel, changing the metamodel requires also changing the transformation rules. Different from the transformation rules in Section 2, the rules are now defined with constraints as shown in Table 2. This implies that we need to use constraint-aware model transformations as detailed in [21]. We use colors to denote whether an element is deleted (red), added (green) or not changed (black). The rules are almost the same as in Section 2; the difference is that the structures representing state information are replaced with annotations.

5.2 Annotation Encoding

Annotations can be encoded as normal node and edge signatures using Alloy primitive type *Int* with minor changes as follows:

```
1 sig S^E_AG{src:one N, trg:one Int}{trg>=0 and trg<n} //n is the number of
     annotations contained within the group
2 sig Graph{nodes:set S_N1+...+S_Nm+Int, edges:set S_E1+...+S_En+S^E_AG}
```

1. In each annotation group AG, each annotation is indexed with a unique number. For each AG, an edge signature S^E_{AG} is created, where its *src* field is N, the applicable type of the group; while the *trg* field is the corresponding signature index. The edge signature encodes that a node is marked with an annotation from the AG.
2. The signature *Graph* contains implicit elements Int and S^E_{AG} besides explicit elements $Node_i$ and $Edge_j$, as shown in line 2 in the listing above.

The three *AG* in the metamodel in Fig. 3 are encoded as follows:

```
sig AP_Flag{src:one P, trg:one Int} {trg>=0 and trg<2}
sig AP_At{src:one P, trg:one Int} {trg>=0 and trg<4}
sig AP_IsActive{src:one P, trg:one Int} {trg>=0 and trg<3}
sig Graph{nodes:set P+R+T+Int, edges:set PR+TP+TR+AP_Flag+AP_At+AP_IsActive}
```

6 Experiments and Results

In this section, we present experiments to study how the three techniques tackle the scalability problem and improve the performance. The first experiment shows how the change of encoding affects the scalability. The last two experiments show the effect of each optimization separately (see Table 3). All the experiments perform conformance verification and are executed on an Intel®Core™i5-2410M @2.30GHz*4 machine with 4GB RAM. The left column shows the scope, and each group of 3 columns shows the result of applying the techniques. Since the Alloy Analyzer uses SAT solver, we present the time cost for translation from Alloy specification to a SAT problem (TR), the verification time to solve the SAT problem (VE) and the total time (TO). Note that the table shows the total time for verifying all the constraints.

The results show that after changing the encoding procedure the approach scales better since we can verify the transformation system for larger scopes (even larger than 14). The results also indicate that the verification with the new encoding procedure is time consuming with large scopes; e.g. with scope 14 the TR is 33.8 minutes while the VE of is 8.5 minutes for all the 28 constraints.

With the decomposition technique, VE decreases to 30 seconds and the TR is around 3 minutes for the 28 constraints. The annotation technique shows even better performance: since the annotation approach reduces the constraints in the metamodel from 28 to 10, within scope 14, the TR reduces to 10125 ms, while the VE decreases to 16041ms for the 10 constraints.

Table 3: Summary of verification performances (unit:millisecond)

Scope	Old Encoding			New Encoding			Decomposition			Annotation		
	TR	VE	TO	TR	VE	TO	TR	VE	TO	TR	VE	TO
3	6235	3084	9319	4489	1615	6104	4357	1831	6188	1497	599	2096
4	9655	5236	14891	6530	1679	8209	3535	1552	5087	1786	944	2730
5				13194	2891	16085	3506	2365	5871	2320	751	3071
6				29257	5809	35066	4189	4668	8857	2357	980	3337
7				60861	9974	70835	5652	7769	13421	2703	1056	3759
8				114951	21960	136911	7173	14622	21795	2906	1544	4450
9				209371	52176	261547	10066	41465	51531	3331	2421	5752
10				357734	90784	448518	12576	45531	58107	3792	5866	9658
11				581151	162532	743683	15246	92928	108174	5272	5081	10353
12				910631	307525	1218156	19387	71090	90477	6766	9291	16057
13				1408900	307081	1715981	24797	243451	268248	8123	13673	21796
14				2030142	509732	2539874	30336	180909	211245	10125	16041	26166

Note that although the decomposition technique does not lead to as good performance as application of annotation, it is more generic in the sense that it could be applied to any model transformation system. But the annotation approach can only be used when state information is present in the system.

7 Related Work

The literature related to the verification of model transformation systems is becoming abundant. Some works propose verification approaches specialized for specific languages. For example, verification of DSLTrans [6] and ATL transformation [17] are studied in [18] and [7], respectively. Both languages are terminating and confluent by nature. Because of this, both works verify model syntax relations [2], which verify whether certain elements of the source model have been transformed into elements of the target model. Different from these works, our approach applies for verification of graph-based model transformations and properties without being restricted to a certain transformation language.

GROOVE [14] verifies graph-based model transformations using model checking. In this approach, the initial state must be given and it works only for finite state spaces. In addition, it encounters the state space explosion problem which is well-known in model checking. Simone et al. [8] uses relational structures to encode graph grammars and FOL to encode graph transformations. In this way, they provide a formal verification framework to reason about graph grammars using mathematical induction, which needs mathematical knowledge. Guerra et al. [15] proposed an automatic verification approach based on visual contract. In this approach, test input models are generated by a constraint solver and properties specified as contracts are verified by their algorithm on those input models. Similarly, our work use constraint solver techniques and is input independent [2]. By contrast, the approaches with constraint solvers are incomplete in the sense that the properties are verified within a certain coverage of instances. However, it can be used to quickly find bugs in a system and provide feedbacks about which parts of the system cause the errors. Furthermore, we do not build the whole state space. This avoids the state space explosion problem which the model checking approaches encounter. There are also several previous works [3,4,9] for verification of transformation systems with Alloy. But they only give encoding of specific examples, without offering a general encoding procedure.

8 Conclusion and Future Work

In this paper we have presented an extension of our previous work [22] to tackle scalability and performance issues by employing three techniques: changing the encoding procedure, decomposing patterns into sub-patterns and applying annotation to represent state information. The experimental results show that the techniques improve the scalability and performance (i.e. translation and verification time). In the future we will examine how the approach can be used to verify other properties like liveness, deadlock, etc. Furthermore, more cases will be examined to study the effects of the optimization techniques and evaluate how the approach performs for more complex model transformation systems. Currently, we encode direct model transformations applying only one rule, but in future we would like to encode model transformations which apply rules concurrently (composition of direct model transformations).

References

1. Alloy. *Project Web Site*. `http://alloy.mit.edu/community/`.
2. M. Amrani, L. Lúcio, G. Selim, B. Combemale, J. Dingel, H. Vangheluwe, Y. L. Traon, and J. R. Cordy. J.r.: A tridimensional approach for studying the formal verification of model transformations. In *Proc. of VOLT*, 2012.
3. K. Anastasakis, B. Bordbar, and J. M. Küster. Analysis of Model Transformations via Alloy. In *Proc. of MoDeVVa*, 2007.
4. L. Baresi and P. Spoletini. On the Use of Alloy to Analyze Graph Transformation Systems. In *Proc. of ICGT*, 2006.
5. M. Barr and C. Wells. *Category Theory for Computing Science (2^{nd} Edition)*. Prentice-Hall, Inc., 1995.
6. B. Barroca, L. Lucio, V. Amaral, R. Félix, and V. Sousa. DSLTrans: A Turing Incomplete Transformation Language. In *Proc. of SLE*, 2010.
7. F. Büttner, M. Egea, and J. Cabot. On verifying ATL transformations using off-the-shelf SMT solvers. In *ACM/IEEE MODELS 2012*, LNCS, 2012.
8. S. A. da Costa and L. Ribeiro. Verification of graph grammars using a logical approach. *Science of Computer Programming*, 77:480–504, 2012.
9. Z. Demirezen, M. Mernik, J. Gray, and B. Bryant. Verification of DSMLs Using Graph Transformation: A Case Study with Alloy. In *Proc. of MoDeVVa*, 2009.
10. E. W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Commun. ACM*, 1965.
11. Z. Diskin. *Encyclopedia of Database Technologies and Applications*, chapter Mathematics of Generic Specifications for Model Management I and II. Information Science Reference, 2005.
12. Z. Diskin, B. Kadish, F. Piessens, and M. Johnson. Universal Arrow Foundations for Visual Modeling. In *Diagrams 2000: 1^{st} International Conference on Diagrammatic Representation and Inference*, 2000.
13. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer. Springer-Verlag New York, Inc., 2006.
14. A. H. Ghamarian, M. J. de Mol, A. Rensink, E. Zambon, and M. V. Zimakova. Modelling and analysis using GROOVE. *International journal on software tools for technology transfer*, 2012.
15. E. Guerra, J. de Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger. Automated Verification of Model Transformations Based on Visual Contracts. *Autom. Softw. Eng.*, 20(1), 2013.
16. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
17. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2), 2008.
18. J. D. S. M. Levi Lúcio and H. Vangheluwe. A technique for symbolically verifying properties of graph-based model transformations. Technical Report SCOCS-TR-2012, McGill University, 2013.
19. A. Rutle. *Diagram Predicate Framework: A Formal Approach to MDE*. PhD thesis, Department of Informatics, University of Bergen, 2010.
20. A. Rutle, W. MacCaull, H. Wang, and Y. Lamo. A metamodelling approach to behavioural modelling. In *Proc. of BM-FA*, 2012.
21. A. Rutle, A. Rossini, Y. Lamo, and U. Wolter. A formal approach to the specification and transformation of constraints in MDE. *JLAP*, 2012.
22. X. Wang, Y. Lamo, and F. Büttner. Verification of graph-based model transformation using Alloy. In *Proc. of GTVMT*, 2014.

A Viewpoint-Based Approach for Formal Safety & Security Assessment of System Architectures

Julien Brunel[1], David Chemouil[1], Laurent Rioux[2],
Mohamed Bakkali[3], and Frédérique Vallée[3]

[1] Onera/DTIM
F-31055 Toulouse
firstname.lastname@onera.fr
[2] Thales Research & Technology
F-91767 Palaiseau, France
laurent.rioux@thalesgroup.com
[3] All4Tec F-53001 Laval, France
firstname.lastname@all4Tec.net

Abstract. We propose an model-based approach to address safety and security assessment of a system architecture. We present an integrated process where system engineers design the model of the system architecture, safety and security engineers specify the propagation of failures and attacks inside each component of the architecture using their dedicated tool. They also define the failure modes that have to be merged from both disciplines. The underlying analyses are then performed using Alloy. We instantiate this approach with the system engineering tool Melody from Thales, and the risk analysis supporting tool Safety Architect from All4Tec. We illustrate this work on a system that implements a landing approach of an aircraft.

1 Introduction

Safety and security are commonly identified disciplines in system and software engineering. In critical embedded system engineering, the fact to spend a lot of effort in safety engineering is a common practice, in particular because these systems generally need to be certified. Standards specify a complete and precise safety process to follow in order to be certified. More recently, architects have begun considering *security* with more attention. Indeed malicious attacks on the system may cause failures and catastrophic events. So, there is a need to not only assess the safety properties but also the security properties of critical embedded systems to create a dependable system architecture. However, the literature shows the difficulties to combine these disciplines in engineering [4].

In [3], we showed that it is possible to assess some properties of a critical embedded system architecture by using the lightweight formal language Alloy. This is a promising solution but industrial companies may not accept to require safety and security engineers to create and maintain Alloy formal models. One problem is then to rely on Alloy for formal analysis while hiding it to end-users. On the other hand, safety standards are also evolving to encourage the use of design models (MBSE, Model based

System Engineering), as well as formal techniques and tools to assess properties inside these models (MBSA, Model based System Assessment).

This article introduces a proposal for a viewpoint-based approach to integrate formal assessment with Alloy in a modelling context. Remark that we focus on the feasibility of the whole approach rather than on viewpoint-based engineering *per se* (*e.g.* overall consistency, name management, abstraction layers...) hence our approach is quite simple w.r.t. the current state-of-the-art on viewpoints [5].

Now since safety engineering and security engineering rely on specific tools, we propose a solution with 3 viewpoints:

- A design layer where system architects design the system architecture (here with the Thales in-house tool called "Melody");
- A safety/security layer where safety engineers and security engineers (based on extensions for security of the "Safety Architect" tool) can model their safety and security properties (dysfunctional and security attack model);
- A third layer which consists in a formal model (here an Alloy model) to assess safety and security properties of the system architecture.

Our approach is currently done by hand as this work is a feasability study, the purpose of which is mainly to focus on viewpoints and formal validation, rather than implementing model transformations which would be rather simple here. Notice that, as of today, the feedback of the assessment results to the system architecture design are still under study and then not addressed in this article.

2 LPV case study

This case-study is concerned by the architecting of a new Thales Avionics aircraft embedded system designed to support an LPV *landing approach*. Localizer Performance with Vertical guidance (LPV) is the highest precision GNSS aviation instrument approach procedure currently available without specialized aircrew training requirements. LPV is designed to provide 16 meter horizontal accuracy and 20 meter vertical accuracy 95 percent of the time. Its architecture is represented by Fig. 2.

We can summarize the behavior of this sub-system as follows. Two Global Navigation Satellite Systems (GPS and GALILEO) send a signal to `SBAS` processing functions. After correlations of both positions information, the `SBAS` sends the aircraft position (lateral and vertical) to two occurrences of the `LPV processing` function. The data produced by `LPV processing` functions are sent to three displays (three occurrences of a function `Acquire`). In each display, a comparison of the data received from `LPV1` and `LPV2` is performed. In case of inconsistency, an alarm is triggered by a function `Monitor`. The crew chooses which of the two LPV processings is used by each display (function `SelectSource`, not represented in Fig 2). Besides, each display receives the data computed by the other two displays. Then, the function `Crosscheck` compares the data of the current display with the data of the two others and resets the current display in case it differs from the other two displays.

This initial architecture was designed taking into account a number of safety requirements; two of these are recalled below.

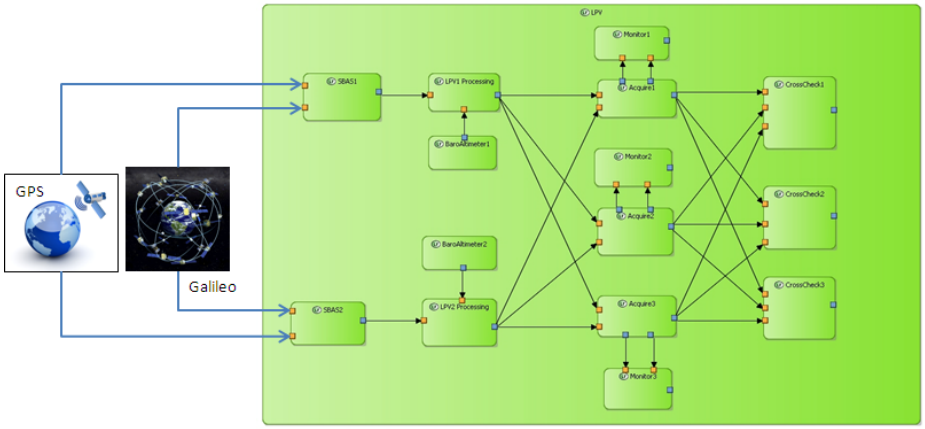


Fig. 1. LPV architecture

Safety 1 *Loss of LPV capability*. No single failure must lead to the loss of LPV capability.

Safety *Misleading information integrity*. The architecture must control the value of the LPV data provided by each calculator and between each screen and find mitigation in case of erroneous data.

We also want to ensure that the above architecture is resilient to a number of malevolent attacks. Any combination of the following attacks has been considered.

Attack 1 One malicious GPS signal (a fake signal that SBAS considers to come from GPS).

Attack 2 One constellation satellite signal is scramble.

Attack 3 The RNAV ground station is neutralized, meaning that no more RNAV signal can be send to the plane.

We will see in Sect. 5 that these requirements are easily expressible (and checked) in Alloy.

3 Model Based Safety & Security Assessment

3.1 Model Based System Engineering (MBSE)

System Engineering of aerospace electronic devices and systems (e.g. avionics, flight or aircraft systems control, mission computers . . .) is submitted to high constraints regarding safety, security, performance, environment, human factors and more; all of these deeply influence systems architecture design and development, and are to be reconciled in a relevant system architecture. The model-based system engineering (MBSE) is an efficient approach to specifying, designing, simulating and validating complex systems. This approach allows errors to be detected as soon as possible in the design process, and thus reduces the overall cost of the product. Uniformity in a system engineering project,

which is by definition multidisciplinary, is achieved by expressing the models in a common modeling language.

Due to its position of large mission-critical systems supplier for aerospace, defense & security markets, THALES invests a lot in system engineering.In particular, THALES has developed its own MBSE method named ARCADIA [11,10]. ARCADIA is based on architecture-centric and model-driven engineering activities, supported by a tool called Melody.

3.2 Model Based Safety Assessment

Model-based safety assessment is nowadays more and more considered in order to improve the safety analysis of complex systems. It relies on the idea that safety assessment activities can follow the design process in a parallel flow using the system functional and physical architectures as a common basis. The system model, either functional or physical, is used to capture the overall architecture and the interactions between its components. This abstract view of the system may be enriched with safety information using dedicated annotations in order to describe possible dysfunctional behaviors.

Safety Architect is a tool achieving risk analysis of complex systems using functional or physical architectures. Safety Architect allows the user to automatically generate the Fault Tree through a “local analysis” (see Fig. 2). The local analysis consists in linking with logical links (“and”, “or”) failure modes of the outputs of each component to the failure modes identified on the component inputs. During the local analysis, the user can also describe the component internal failures effects on its outputs.

In parallel, the user can also identify safety barriers that prevent the development of a single fault up to particular failure mode that could lead to a hazardous event, participating thus to the safety objectives compliance. The user must also define which failure modes of the system outputs have to be considered as hazardous events. These events are the subject of the “global analysis” provided by the tool Safety Architect.

During the global analysis, a dysfunctional simulation of the system is executed by propagating failures along the dataflow dependencies of components and until a hazardous event is reached. The results of this propagation are formulated through Fault Trees, the roots of which are all the previously identified hazardous events.

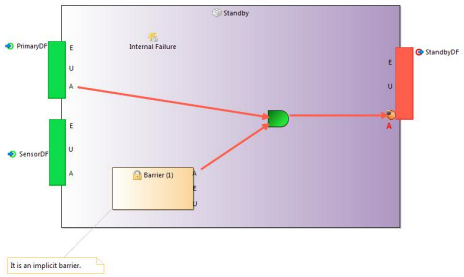


Fig. 2. Example of local analysis

3.3 Formal techniques

Formal techniques can be used to support safety and security assessment. Thanks to their mathematical foundation, they allow to prove some requirements, which provide a better confidence than more classical validation activities such as testing and manual review. A number of verification techniques have been developed over the last decades. They may differ on their expressiveness, their computational complexity and their application domain.

In this work, we have chosen Alloy [6], which is a formal system-modelling language amenable to automatic analyses. Alloy has recently been used in the context of security assessment, for instance to model JVM security constraints [8], access control policies [9], or attacks in crytogrpahic protocols [7]. Besides, we proposed in earlier work a preliminary study of the safety assessment of the LPV system with the study of a few security attacks [2,3].

The AltaRica [1] language, which is widely use for safety assessment, would have been another possible choice. However, we decided to take benefit from the model-based aspect of Alloy and its expressiveness for the specification of the properties to check. Indeed, Alloy allows to define easily the metamodel of the avionic architectures we will analyze instead of encoding them in terms of AltaRica concepts. Moreover, the specification of the properties we want to check are expressed in relational first-order logic with many features adapted to model-based reasoning.

4 Proposed approach for MBS&SA

4.1 Main principles

The approach we propose in this article consists in decoupling the system architecture model from safety & security models. This way, every engineer (be it an architect, a security or a safety engineer) can focus on her concerns solely, with dedicated tools and terminology. As of now, we chose to use two separate models: one for the safety concern and the second for the security concern. The main motivation for this separation is that safety and security domains are quite different in terms of practices, concepts used and wording. As the safety and security models rely on the system architecture model, we extract required information (e.g. functions interactions, ports and their links, data) from the architecture model and we set up initial safety and security models in Safety Architect. Starting from this, safety and security engineers complete their model by adding safety and security dysfunctional behavior. The safety and security models contain two kinds of information: the dysfunctional behavior and the properties (safety or security) to be validated. For us, a safety dysfunctional behavior represents how errors are propagated in the system architecture and a security dysfunctional behavior represent how security attacks are propagated in the system architecture. And the safety and security properties are mainly safety and security requirements that the system architecture must satisfy (e.g. integrity of the output data must be preserved even under specific attacks). Finally, these two models are combined to produce a formal Alloy model containing all the necessary input. Then, the Alloy Analyzer can formally validate the safety and security properties. If a property is violated, the Alloy Analyzer will

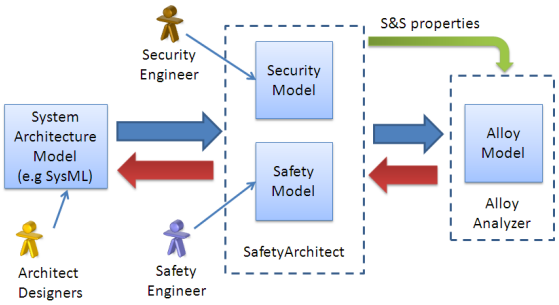


Fig. 3. Proposed approach

show a readable corresponding counter-example. This way, the engineers can identify the best way to correct the architecture to solve this identified issue.

4.2 Melody to Safety Architect

The first model transformation yields an initial Safety Architect model from the system design model. This transformation is trivial as it only reflects the structural part of the architecture. Melody functions are mapped to Safety Architect in functions. Ports give input and output ports, while data links yield data links.

4.3 Safety Architect to Alloy

We now present (a fragment of) the Alloy formalization of the language used in Safety Architect. Essentially, we define sets and relations between them: the former are called *signatures* in Alloy while the latter are described as *fields* inside the said signatures. First, we define a notion of `status` which is a signature the elements of which represent types of failures: `Absent`, `Err` (erroneous) and `Mal` (malicious) while `OK` just represents that no failure happened.

```
enum Status { OK, Err, Abs, Mal }
```

Then, blocks are mapped to functions endowed with possibly-many input and output ports as well as one status which is used to represent the notion of internal failure from Safety Architect:

```
abstract sig Function { input: set IPort, output: set OPort, status: Status }
```

Finally, ports also come with a status and can either be input or output ports. An output port may be connected to many input ports, as expressed by the field `flow`:

```
abstract sig Port { status: Status }
abstract sig IPort extends Port {}
abstract sig OPort extends Port { flow: set IPort }
```

Notice that the notion of internal failure from Safety Architect is mapped to the `status` in `Function`, although other formalizations would have been possible.

Along with these signatures, we have some Alloy *facts* which enforce static invariants on possible instances of this formalization. We do not describe them here as they are rather obvious (*e.g.* a block should have at least one port; if two ports are connected, then they should bear the same value and status...).

5 Case study evaluation

The LPV model is imported into Safety Architect modeler as shown in Fig. 4. The objective is to have either the safety view or the security view or the combination of both views as needed.

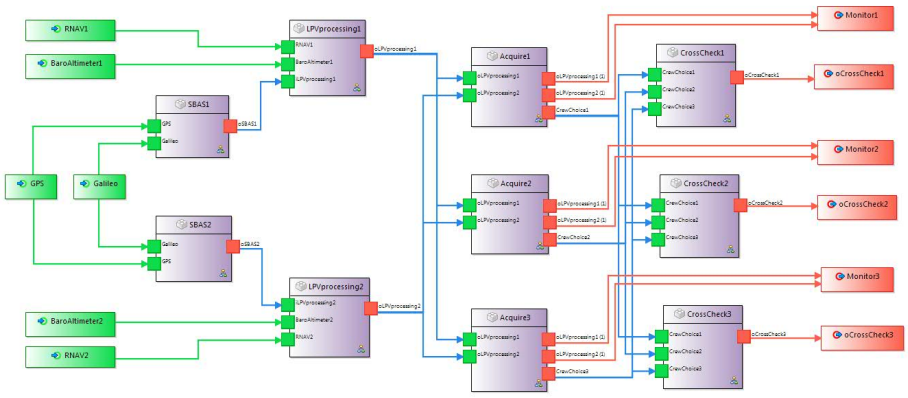


Fig. 4. LPV model in Safety Architect

5.1 Safety model

The safety analyses are based on logical equations using three generic failure modes proposed by Safety Architect on each input (in green) of a block:

A Absent (Absent data while it should be present)
E Erroneous (Non correct supplied data)
U Untimely (Data supplied while it shouldn't be)

If necessary, specific failure modes can also be defined on the input. The Untimely failure mode is defined by default but we did not use it in this case study.

For example we can say for the Safety analysis (Fig. 5, left) that one of the two ways to observe the "Erroneous" failure mode on the output "oLPVprocessing1" is to have the "Erroneous" failure mode on the input "iLPVprocessing1" *and* the "Absent" failure mode on the input "BaroAltimeter1" *or* the "Absent" failure mode on the input "RNAV1".

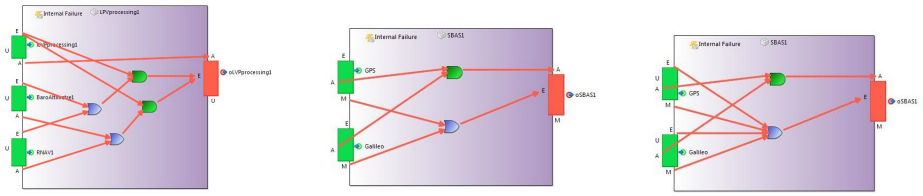


Fig. 5. Safety analysis of the block LPVprocessing1 [left] / Security analysis of the block SBAS1 (Attack 4) [middle] / Combination of the Safety and Security analysis of the block SBAS1 [right]

5.2 Security model

For security analysis, Safety Architect proposes three other generic failure modes on each input of a block:

A Absent (Absent data due to an external attack)
M Malicious (Data injected during an external attack)
E Erroneous (Malicious detected data)

Let us consider for example the security analysis of the block SBAS1 illustrated by Fig. 5 (middle). It covers "attack 4" (an attack combining attack 1 and attack 2 scenarios i.e. when SBAS considers a fake signal coming from GPS and one constellation signal is scrambled). One can see that the "Erroneous" failure mode of the output "oSBAS1" is obtained iff the "Malicious" failure mode on the input "GPS" *or* the "Malicious" failure mode on the input "Galileo" holds.

5.3 Safety and security model

The safety and security model combines both the safety and security views. By default, this is implemented as follows (but the user may modify this discretely depending on domain knowledge):

- Failure modes with the same name are identified;
- The set of logical equations of the resulting model is the union of the sets (of logical equations) of the safety and security views. However, if an equation in the safety view concerns the same output port and failure mode than an equation in the security view, there is only one resulting equation: the disjunction of both equations. The rationale is that we want to keep the two different ways for the output port to propagate the said failure.

An example is shown in Figure 5 (right). Combining the safety and security views in one model allows us to merge the two propagations into a unique propagation. The latter shows the intersection between both views; it also allows the safety or the security engineer to identify which safety or security (or both) failure modes may contribute to the appearance of a Feared Event.

5.4 Alloy code generation

We already presented in Sect. 4.3 how Safety Architect concepts (blocks, ports, failure modes) are translated into Alloy. We now show what is the Alloy representation of (an excerpt of) our case study and how to specify safety and security requirements.

Let us consider the block illustrated in Sect. 5.3 (SBAS1). Firstly, we have to declare it (as a Function) and its three ports.

```
one sig SBAS1 extends Function {}
one sig oSBAS1 extends OPort {}
one sig iGPS_SBAS1, iGalileo_SBAS1 extends IPort {}
```

We then express the connections between (ports of) functions as an Alloy constraint (a conjunction of equality between ports). Then we translate the failure propagation inside the block as Alloy facts as follows.

```
let oSBAS1 = { GPS = Abs and Galileo = Abs implies Lost
               else GMS = Mal or Galileo = Mal implies Err
               else OK }
```

Finally, we can express requirements to check directly as Alloy assertions. Note that from the identification of feared event in the Safety Architect model, we could easily generate patterns of requirements that would express that no single failure leads to this feared event, of that no attack of a certain type lead to this feared event, or that no combination of failure and attack lead to this event, etc.

For instance, the following assertion states that a fake GPS signal (attack 1 described in Sect. 2) has no bad influence on the system (the data sent by the three displays, represented by variables oSelected*i*, are still correct).

```
assert fake-GPS-has-no-bad-influence {
   (all f: Function | f.status=OK and GPS.status=Mal)
      implies oSelected1.status = OK and oSelected2.status = OK
      and oSelected3.status = OK }
```

This requirement can be verified by Alloy Analyzer with the command
`check fake-GPS-has-no-bad-influence`.

We have expressed and checked the safety requirements described in Sect. 2 and the security requirement relative to the attacks described in Sect. 2 in a similar way. It turns out that the system is robust to any single failure and to any simple attack (attack 1, 2 or 3). We also checked the consequences of any combination of two attacks: depending on the considered combination, either the system is robust or an alarm, not represented in this article, is launched. The same conclusion holds for any combination of an attack and a function failure.

6 Conclusion and future work

In this article, we proposed a model-based approach to address safety and security assessment of a system architecture. We proposed a way to make system engineers, safety

engineers and security engineers collaborate in order to perform safety and security assessment in the easiest possible way.

Now we see the feasibility and the interest of this approach, the next step is to implement it. We will need to address classical but important problems, such as the traceability between the Safety Architect models and the Melody model. For instance, after an evolution of the system architecture performed under Melody, we will have to ensure that the failure propagation inside blocks described with Safety Architect does not need to be entirely redefined.

References

1. A. Arnold, G. Point, A. Griffault, and A. Rauzy. The altarica formalism for describing concurrent systems. *Fundamenta Informaticae*, 40(2,3):109–124, Aug. 1999.
2. J. Brunel, D. Chemouil, N. Mélédo, and V. Ibanez. Formal modelling and safety analysis of an avionic functional architecture with alloy. In *Embedded Real Time Software and Systems (ERTSS 2014)*, Toulouse, France, 2014.
3. J. Brunel, L. Rioux, S. Paul, A. Faucogney, and F. Vallée. Formal safety and security assessment of an avionic architecture with alloy. In *Proceedings Third International Workshop on Engineering Safety and Security Systems (ESSS 2014)*, volume 150 of *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, pages 8–19, 2014.
4. D. G. Firesmith. Engineering safety- and security-related requirements for software-intensive systems: tutorial summary. In *International Conference on Software Engineering - Volume 2 (ICSE 2010)*, pages 489–490. ACM Press, 2010.
5. IEEE Architecture Working Group. ISO/IEC/IEEE 42010 Systems and software engineering - Architecture description. The latest edition of the original IEEE Std 1471:2000, Recommended Practice for Architectural Description of Software-intensive Systems, 2011.
6. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
7. A. Lin, M. Bond, and J. Clulow. Modeling partial attacks with alloy. In B. Christianson, B. Crispo, J. Malcolm, and M. Roe, editors, *Security Protocols*, volume 5964 of *Lecture Notes in Computer Science*, pages 20–33. Springer Berlin Heidelberg, 2010.
8. M. Reynolds. Lightweight modeling of java virtual machine security constraints. In M. Frappier, U. Glässer, S. Khurshid, R. Laleau, and S. Reeves, editors, *Abstract State Machines, Alloy, B and Z*, volume 5977 of *Lecture Notes in Computer Science*, pages 146–159. Springer Berlin Heidelberg, 2010.
9. M. Toahchoodee and I. Ray. Using alloy to analyse a spatio-temporal access control model supporting delegation. *Information Security, IET*, 3(3):75–113, Sept 2009.
10. J.-L. Voirin. Method and tools to secure and support collaborative architecting of constrained systems. In *27th Congress of the International Council of the Aeronautical Science (ICAS 2010)*, 2010.
11. J.-L. Voirin and S. Bonnet. Arcadia: Model-based collaboration for system, software and hardware engineering. In *Complex Systems Design & Management, poster workshop (CSD&M 2013)*, 2013.

Modeling Spatial Aspects of Safety-Critical Systems with FocusST

Maria Spichkova[1], Jan Olaf Blech[1], Peter Herrmann[2], and Heinz Schmidt[1]

[1] RMIT University, Melbourne, Australia
{maria.spichkova, janolaf.blech, heinz.schmidt}@rmit.edu.au
[2] Norwegian University of Science and Technology (NTNU), Trondheim, Norway
herrmann@item.ntnu.no

Abstract. This paper presents an approach for modeling and verification of components controlling behaviour of safety-critical systems in their physical environment. In particular, we introduce the modeling language FocusST that is centred on specifying time and space aspects. Verifications can be carried out using the interactive semi-automatic proof assistant Isabelle. The approach is exemplified by means of a railway system scenario.

1 Introduction

Many safety-critical systems (SCSs) consist of mobile units autonomously moving in their physical environment. Modeling such systems requires not only the definition of the software part but also a specification of interactions with the physical environment. In consequence, the models need to capture timing and spatial aspects that should provide a basis for formal verification of safety properties. In most cases, however, we do not need the whole representation of an SCS but only those parts relevant to a concrete purpose. Thus, an appropriate model should give an overview of core system properties and allow an effective inconsistencies funding, reducing modeling and verification effort.

For modeling SCSs suitably, it is essential to have a well developed theory covering real-time and space requirements since mistreating or excluding them can lead to specification errors due to difficulties of choosing a correct abstraction. Moreover, in many cases reasoning about time to represent a real-time system makes the specification more readable (in comparison to an untimed representation), simplifies the argumentation about its properties, and gives a formal basis for verification. A suitable representation of SCSs should also make it possible to model information flow not only in time but also in space, because the spatial aspect may influence the delays of interactions between subcomponents of the system as well as between the system and the environment. This point is important for cost reduction of interoperability testing at the integration phase of the development process. Further, for a versatile application of a notation, the selection of a suitable space-time coordinate system should be relatively free.

The modeling language that we use in our approach is FocusST. It allows us to create concise but easily understandable specifications and is appropriate

for application of the specification and proof methodology presented in [19, 25]. This methodology allows writing specifications in a way that carrying out proofs is quite simple and scalable to practical problems. In particular, a specification of an SCS can be translated to a Higher-Order Logic and verified by the interactive semi-automatic theorem prover Isabelle [17] also applying its component Sledgehammer [4]. Sledgehammer employs resolution based first-order automatic theorem provers (ATPs) and satisfiability modulo theories (SMT) solvers to discharge goals arising in interactive proofs. Another advantage is a well-developed theory of composition as well as the representation of processes within a system [20]. The collection of FOCUSST operators over timing aspects and their properties specified and verified using the theorem prover Isabelle is presented in the Archive of Formal Proofs [21]. In this work we focus on modeling of spatial aspects.

Related Work: One of the most well-established models for the specification and verification of real-time system design is timed automata, introduced by Alur and Dill [1, 2]. A timed automaton is a finite automaton extended by real valued clocks that are applied to measure the time elapsed since certain events occurred. The clocks are used in so-called clock invariants that restrict the time, a timed automaton may rest in a particular state without executing certain transitions. Timed automata assume perfect continuity of clocks which may not suit the purposes of the work presented here, especially if we deal with an embedded system with instantaneous reaction times. Furthermore, they do not prevent Zeno runs [12], i.e., executing an infinite number of transitions in a finite period of time. To solve this, the idea of robust model checking was introduced by Puri [18] and revised in other approaches, e.g., [8]. In this paper, we suggest another solution: We use asynchronous channels between timed automata and argue about possibly infinite *message sequences* towards an automaton at some time interval. This can be represented by using an infinite sequence of finite time intervals as input for a timed automaton. Any timed transition system can be discretised without loss of generality [14]. For this reason, we apply a discrete model of time where any granularity defining the concrete meaning of a time interval according to the system requirements can be used. We can even switch from one time granularity to another using predefined operators. A great advantage of the proceeding is that it excludes Zeno runs.

Related work regarding spatial aspects has been done with respect to logic and tools. A process algebra like formalism for describing and reasoning about spatial behavior has been introduced in [10, 11]. Process algebras come with a clear and formal semantics definition and are aimed towards the specification of highly parallel systems. Here, disjoint logical spaces are represented in terms of expressions by bracketing structures and carry or exchange concurrent processes. Results on spatial interpretations can be found in [15]. Many aspects of spatial logic are in general undecidable. A quantifier-free rational fragment of ambient logic (corresponding to regular language constraints), however, has been shown to be decidable in [26]. Work on spatial model checking by ourselves is presented in [6, 7]. Furthermore, this approach was coupled with the model-based

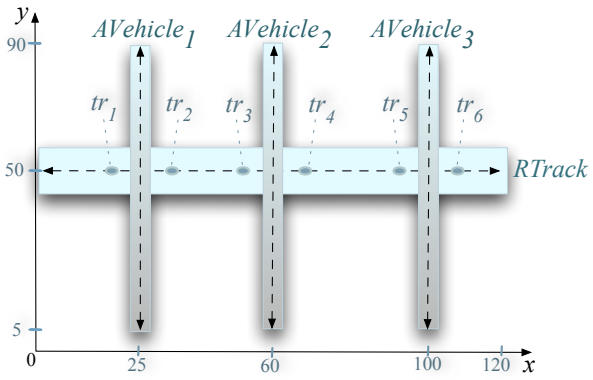


Fig. 1. Automatic transport system

engineering technique Reactive Blocks [16] such that reactive systems (e.g., SCS controllers) can be developed using models and be checked for spatial properties before generating executable code [13].

Scenario: Due to the well specified degrees of freedom enforced by rail tracks, trains have been a popular target for verification work (see, e.g., [3]). This makes them also an appropriate target to introduce the main idea of modeling and verification of spatial aspects. Here, we present a small example from this area that is depicted in Fig. 1. A train *RTrack* shuttles on a rail track that is crossed by three roads. On each road, an autonomous vehicle *AVehicle* is operating. All four mobile units are characterized by a number of constraints on their location, speed, movement direction, etc. To avoid collisions, the train sends a *wait* signal to the respective *AVehicle* while passing one of the corresponding critical points tr_i close to the crossings ($1 \leq i \leq 6$). If a critical point is ahead of a crossing in the direction of the train, the *wait* signal expresses a time interval, for which the *AVehicle* has to stop if it is heading towards the crossing and is far enough to stop in due time. The time interval may depend on the speed of the train giving it sufficient time to pass the crossing. If a critical point is behind a crossing, *wait* contains value 0 indicating that the vehicle may immediately continue moving. The fact that an *AVehicle* is only stopped for a certain time interval at most, provides a challenge on the space-related behavior since we have to guarantee that the train already left the crossing when the time interval passed. This poses the following questions: How should we model spatial properties of this system in a readable way? Does this modeling technique allow formal verification of safety properties? Is this model also appropriate to specify and verify a large number of components, e.g., for the case that we have not three but one thousand *AVehicle* components? In this paper we will answer these questions by presenting our modeling approach in FOCUSST.

2 Spatial Aspects in FocusST

The FocusST language was inspired by Focus [9], a framework for formal specification and development of interactive systems. In both languages, specifications are based on the notion of *streams*. However, in the original Focus input and output streams of a component are mappings of natural numbers $\mathbb{N}$ to single messages,whereas a FocusST stream is a mapping from $\mathbb{N}$ to lists of messages within the corresponding time intervals. Moreover, the syntax of FocusST is particularly devoted to specify spatial (S) and timing (T) aspects in a comprehensible fashion, which is the reason to extend the name of the language by ST. The FocusST specification layout also differs from the original one: it is based on human factor analysis within formal methods [22, 23].

We specify every component using assumption-guarantee-structured templates. This helps avoiding the omission of unnecessary assumptions about the system's environment since a specified component is required to fulfil the guarantee only if its environment behaves in accordance with the assumption. In a component model, one often has transitions with local variables that are not changed. Also, outputs are often not produced, e.g., when a component gets no input or some preconditions necessary to produce a nonempty output are violated. In many formal languages this kind of invariability has to be defined explicitly in order to avoid underspecified component specifications. To make our formal language better understandable for programmers, we use in FocusST so-called *implicit else-case* constructs. That means, if a variable is not listed in the guarantee part of a transition, it implicitly keeps its current value. An output stream not mentioned in a transition will be empty. Further, we do not require to introduce auxiliary variables explicitly: The data type of a not introduced variable is universally quantified in the specification such that it can be used with any data value.

The FocusST specifications are a special form of timed automata that we name *Timed State Transition Diagrams* (TSTDs). A TSTD can be described in both diagram and textual form. For easier argumentation, we can further represent it by a special kind of tables including a number of new operators that work on time intervals. For a real-time system S with a syntactic interface $(I_S \rhd O_S)$, where I_S and O_S are sets of timed input and output streams respectively, a TSTD corresponds to a tuple $(\mathit{State}, \mathit{state}_0, I_S, O_S, \rightarrow)$, in which State is a set of states, $\mathit{state}_0 \in \mathit{State}$ is the initial state, and $\rightarrow \subseteq (\mathit{State} \times I_S \times \mathit{State} \times O_S)$ represents the transition function of the TSTD.

An input action for a TSTD is the set of current time intervals of the input streams of the system, while the output action is the set of corresponding time intervals of the output streams of the system. Focus distinguishes between *weak causal systems* and *strong causal systems* (see [9]). In the former case, the output must be produced within the same time interval the input is consumed while in the latter one the output has to be produced within a delay of at least one time unit. The exact delay needs to be defined according to the timing requirements on the specified system.

Spatial Aspects: In addition to the representation of timing properties in the language, we define a special type of components specifying real objects that can physically change their location in space, so-called *sp-objects*. Each sp-object is associated with three special variables storing its current *location* (i.e., central point of the object), *speed* and *direction* of movement. For simplicity, the variable *speed* is defined over the set of natural numbers $\mathbb{N}$, while *location* is of type *Space* and defines a coordinate having two or tree dimensions according to the system's needs. In our two-dimensional example, *Space* is a tuple of two Cartesian coordinates *xx* and *yy*. Finally, *direction* is defined over the type *Directions* $= \{0, \ldots, 359\}$ which represents the angle in the Cartesian coordinate system. In comparison to the local variables declared within components, these three variables are global and can be used to specify physical interaction of components in a system.

A system model may be constrained by restricting the directions and speed of an sp-object. This allows us to verify whether the specified behaviour excludes the possibility that the object enters restricted areas during time intervals marked as dangerous, e.g., collisions with other sp-objects.

FOCUSST *specification:* Figure 2 depicts the textual representation of the sp-object specification pattern for the component *AVehicle* introduced in the scenario section of the introduction. This component is strong causal with a delay of one time unit, and has the three input channels *wait* and *tSpeed* of type $\mathbb{N}$ as well as *tDir* of type *Directions* declared in the interface part of the specification using label "in". The ports *tSpeed* and *tDir* are used to notify changes of the target speed and the target direction of the object. If *AVehicle* is too close to a potential obstacle, e.g., the crossing with *RTrack*, it is signalled via the *wait* port to stop for a number of time units. Thereafter it continues moving with the previous speed. Label "out" defines the output channel *resp* of type *Event* that consists a single element *event* used to signal the start of motion by the vehicle.

Let us name some of the operators used to specify time intervals in our streams: $\langle\rangle$ denotes an empty list, i.e., a single time interval without any events, and $\langle x \rangle$ a list consisting of the element x; $\mathsf{ft}.l$ describes the first element of a list l; s^i represents the ith time interval of the stream s.

Empty brackets after the component's name mean that *AVehicle* does not have any parameters. The component uses the local variable *timer* referring to the current timer value (the special value 0 means that the timer is not active while 1 indicates that it has to time out). By *lspeed*, we store the speed the object carried before needing to stop. The variable *timer* is initially set to be 0 while the initial value of *lspeed* is not specified.

The keyword "asm" lists the assumption, *AVehicle* demands from its environment, i.e., specified using the FOCUSST operator msg_1, at most one message is received via each of the ports *wait*, *tSpeed* and *tDir* at any time interval.

The section "gar" contains the transitions and other formulas. Here, variable settings before executing (i.e. at some time interval t) a transition are marked by simple variable identifiers, e.g., *timer*, while the operator $'$ refers to their setting afterwards, e.g., *timer*$'$ denotes the value of the timer variable at the time

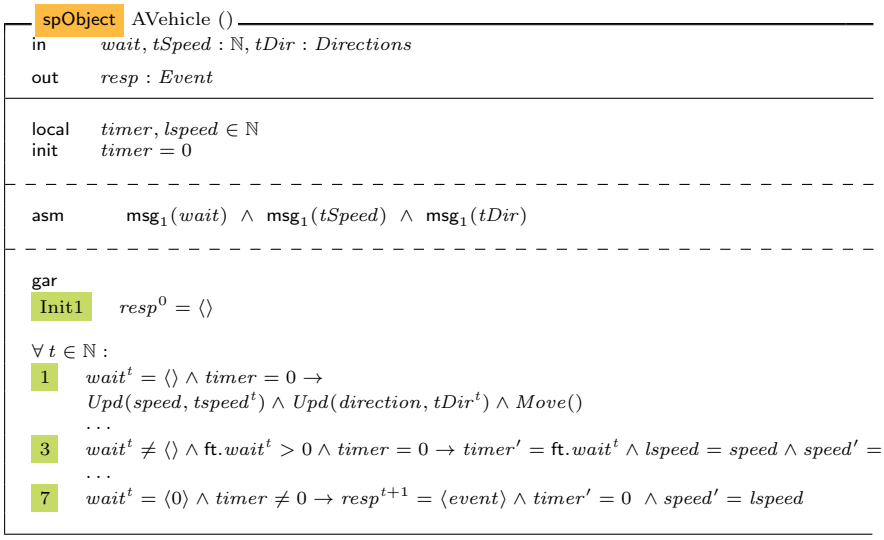


Fig. 2. Textual representation of specification pattern *AVehicle*

interval $t+1$. The initial condition Init1 specifies that the output channel *resp* is empty at the beginning. Formula 1 models a transition that if *AVehicle* does not receive a wait-signal at the time interval t and its timer is inactive, the component moves according its target direction and speed values. This is expressed by the function *Move*() that updates the value of the variable *location*. The function *Upd* defines the updates of the variables *speed* and *direction* according to the events in the input streams *tSpeed* and *tDir* at time interval t. Moreover, due to the implicit else-case construct discussed above, the timer does not change its value and the output stream *resp* is an empty list. Formula 3 specifies the start of the timer if *RTrack* approaches the crossing. In this case, *AVehicle* receives a number $k > 0$ via the port *wait*. The timer is started by setting its variable to k while the vehicle is stopped ($speed' = 0$) and the previous speed is stored in the auxiliary variable *lspeed*. Transition 7 specifies that if *AVehicle* receives 0 via the port *wait* (i.e., it does not need to wait for the *RTrack* any more) while its timer is active, the timer will be set off and, at the next time interval, the *event*-signal will be sent indicating that the vehicle resumes moving again into its target direction. We omit here the rest of the specification due to lack of space.

Fig. 3 shows the diagrammatic version of the TSTD for *AVehicle*. It contains all transitions that we label with the same numbers as in the textual representation. To increase readability of the graphical representation, we distinguish three types of the transition labels by coloured representation: Inputs and constraints on the current local variables' values are marked blue, outputs and changes of

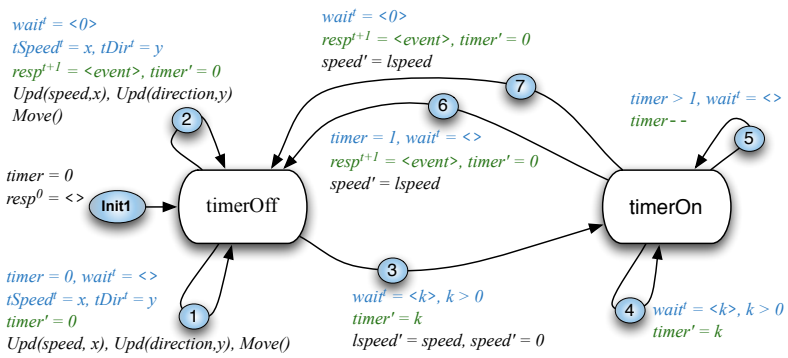


Fig. 3. Timed State Transition Diagram for *AVehicle*

general (non-spatial) local variables' values green, and changes of spacial aspects black.

3 Verification Constraints on Spatial Aspects

We can restrict directions and speed of an sp-object by adding constraints. Predicates are associated with every component. Per default, they are specified as true but can be restricted to represent precise bounds of a component. To calculate whether a collision of sp-objects is possible, we assign to each sp-object a global constant *rad* that describes radius of the maximal space the object can "cover" in the worst case. The maximal space the object can occupy at the time t, is denoted by the variable *rzone* of the type *Zone*. This variable is defined as a tuple $(minX, minY, maxX, maxY)$ of natural numbers which are calculated according to the values of *speed*, *location* and *rad*. The *rad* value of a composite component is defined by analysing which space its subcomponents can occupy in the worst case: $S.rad = max(WCX, WCY)/2$ with WCX and WCY being the maximum extensions of all of the subcomponents of S in direction x resp. y.

We represent the set of all the components' constraints by a table, to increase readability and to check schematically whether constraints on a *composed component* correspond to the constraints on its subcomponents, e.g.,

$$\forall\, S,\, C : C \in subcomp(S) \rightarrow$$
$$(S.rzone.minX \leq S.C.rzone.minX \wedge S.rzone.minY \leq S.C.rzone.minY) \wedge$$
$$(S.rzone.maxX \geq S.C.rzone.maxX \wedge S.rzone.maxY \geq S.C.rzone.maxY)$$

$$\forall\, k,\, S,\, C : C \in subcomp(S) \rightarrow$$
$$(k \leq S.rzone.minX \rightarrow (k + S.C.rad) \leq S.C.location.xx)$$

To analyse spatial properties, we need to specify the rules how the locations of the objects can change over time. The location of the sp-object C at the

Table 1. Set of the spatial constraints

Component	*rad*	*locationRestr*	*speedRestr*	*directionRestr*
$AVehicle_1$	2	$location.xx = 25$	$speed < 10$	$direction = 90 \vee direction = 270$
$AVehicle_2$	2	$location.xx = 60$	$speed < 15$	$direction = 90 \vee direction = 270$
$AVehicle_3$	2	$location.xx = 100$	$speed < 20$	$direction = 90 \vee direction = 270$
$RTrack$	4	$location.yy = 50$		$direction = 0 \vee direction = 180$

beginning of the next time interval can be computed from its speed, direction of movement and the current location. In particular, we specify a *trajectory* of the object during a time interval t to be a straight line, thus, it can be described by the coordinates of two locations, at the beginning of the current and the next time interval: $C.tr = [C.location, C.location']$. Then, we can describe the space where the object C can be *during* the time interval t (let denote it $C.rzoneInterval$) by a set of coordinates:

$$\{(a, b) \mid \exists (a_1, b_1) \in C.tr \wedge a_1 - C.rad \leq a \leq a_1 + C.rad \wedge b_1 - C.rad \leq b \leq b_1 + C.rad\}$$

If $C.speed = 0$ holds, the component C does not move (i.e., $C.location' = C.location$) such that $C.rzoneInterval$ describes the *rzone* space in this case.

Restrictions on location, speed and direction can be specified both pointwise and by using minimum and maximum limits, where a variable can have any value within the defined interval. Let us explain this by using the scenario introduced in Sect. 1. Its spatial contraints are listed in Tab. 1. It is easy to see from this table and Fig. 1 that the four mobile units can occupy 100 units on the x coordinate and 85 units on the y coordinate. Thus, $S.rad$ is assigned with the value 50. By defining an additional constraint for the space that can be used by the overall example system S, we implicitly restrict the corresponding constraints of its components, e.g.,

$$0 \leq S.rzone.minX \wedge S.rzone.maxX \leq 120 \wedge$$
$$5 \leq S.rzone.minY \wedge S.rzone.maxY \leq 90$$

implies that for the component $AVehicle_1$ holds not only $location.xx = 25$ but also $7 \leq location.yy \leq 88$ (taking its own value $rad = 2$ into account). Representing the spatial aspect of a component as a pair of coordinates and a radius, we specify possible collisions between two objects C_1 and C_2 during the time interval t by $PCollision^t(C_1, C_2)$. Thus, an important property for this system is that collisions between the *RTrack* and *AVehicle* components are excluded, i.e. for all $t \in \mathbb{N}$ and $i, j \in \{1, 2, 3\}$ the following holds

$$\neg PCollision^t(RTrack, AVehicle_i) \wedge \neg PCollision^t(AVehicle_i, AVehicle_j)$$

Since in our example, the *AVehicle* objects move on parallel roads, the property on the right side holds trivially. If a *possible* collision is detected, the corresponding case should be analysed carefully both on an abstract (logical) and

on a physical level: Due to our overapproximation of space, not every situation labelled as possibly dangerous on the abstract level indeed corresponds to a real physical collision but, on the other side, any real physical collision must be detectable on the abstract level.

We have used the interactive theorem prover Isabelle/HOL [17] to analyse whether collisions between the *RTrack* and *AVehicle* components are excluded. For example, direction and location constraints together with behaviour specifications imply that the mobile units can collide, if we underspecify the coordinates of the critical points $tr_1, \ldots, tr_6$ as well as the initial locations.

Due to similarity of specifications of separate components, the model of the presented system is scalable not only for specification but also for verification purposes. Even if we have not three but thousand *AVehicle* components, proofs of their spatial behavioural properties can be reused or generated.

4 Conclusions

This paper presents the FOCUSST approach for modeling and verification of safety-critical systems using specifications based on time intervals and spatial aspects. Several features have been demonstrated using an example system based on interacting autonomous vehicles. We focus on timing and spatial aspects as well as readability of the specifications and ease of verification of core properties. For the proofs, we have applied the interactive semi-automatic proof assistant Isabelle.

Our future research direction comprises work on the modeling levels for SCSs, that reflect the idea of remote integration/interoperability testing in a virtual environment [5, 24] as well as automatisation of proof generation for spatial behavioural properties from the system model. Moreover, we want to combine this verification technique with the modeled-based engineering tool Reactive Blocks [16] to facilitate the practical development of the control software for space-aware SCSs.

References

1. R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.
2. R. Alur and P. Madhusudan. Decision Problems for Timed Automata: A Survey. In *SFM*, pp. 1–24, 2004.
3. P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier. Météor: A Successful Application of B in a Large Project. *Formal Methods (FM'99)*, vol. 1708 of *LNCS*, Springer, 1999.
4. J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT Solvers. *Journal of Automated Reasoning* 51(1):109–128, 2013
5. J. O. Blech, M. Spichkova, I. Peake, H. Schmidt. Cyber-Virtual Systems: Simulation, Validation & Visualization. In *9th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2014)*, 2014.

6. J. O. Blech and H. Schmidt. BeSpaceD: Towards a Tool Framework and Methodology for the Specification and Verification of Spatial Behavior of Distributed Software Component Systems. In *arXiv.org*, `http://arxiv.org/abs/1404.3537`, 2014.
7. J. O. Blech and H. Schmidt. Towards Modeling and Checking the Spatial and Interaction Behavior of Widely Distributed Systems. In *Improving Systems and Software Engineering Conference*, 2013.
8. P. Bouyer, N. Markey, and O. Sankur. Robust Model-checking of Timed Automata via Pumping in Channel Machines. *Formal Modeling and Analysis of Timed Systems*, Springer, 2011.
9. M. Broy and K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.
10. L. Caires and L. Cardelli. A Spatial Logic for Concurrency (Part I). *Information and Computation*, 186(2), 2003.
11. L. Caires and L. Cardelli. A Spatial Logic for Concurrency (Part II). *Theoretical Computer Science*, 322(3):517-565, 2004.
12. R. Gómez and H. Bowman. Efficient Detection of Zeno Runs in Timed Automata. *Formal Modeling and Analysis of Timed Systems*, Springer, 2007.
13. F. Han, J. O. Blech, P. Herrmann, and H. Schmidt. Towards Verifying Safety Properties of Real-Time Probability Systems. In *Formal Engineering approaches to Software Components and Architectures (FESCA)*, EPTCS, 2014.
14. T. Henzinger, Z. Manna, and A. Pnueli. What Good are Digital Clocks? In *Colloq. on Automata, Languages and Programming*, pp. 545–558. Springer, 1992.
15. D. Hirschkoff, É. Lozes, D. Sangiorgi. Minimality Results for the Spatial Logics. In *Foundations of Software Technology and Theoretical Computer Science*, LNCS 2914, Springer, 2003.
16. F. A. Kraemer, V. Slåtten and P. Herrmann. Tool Support for the Rapid Composition, Analysis and Implementation of Reactive Services. *Journal of Systems and Software*, 82(12):2068–2080, 2009.
17. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283, Springer, 2002.
18. A. Puri. Dynamical Properties of Timed Automata. *Discrete Event Dynamic Systems*, 10(1-2):87–113, 2000.
19. M. Spichkova. *Specification and Seamless Verification of Embedded Real-Time Systems: FOCUS on Isabelle*. PhD thesis, TU München, 2007.
20. M. Spichkova. Focus on Processes. Tech. Report TUM-I1115, TU München, 2011.
21. M. Spichkova. Stream Processing Components: Isabelle/HOL Formalisation and Case Studies. In *Archive of Formal Proofs*, ISSN 2150-914x, 2013.
22. M. Spichkova. Human Factors of Formal Methods. In *IADIS Interfaces and Human Computer Interaction (IHCI)*. 2012.
23. M. Spichkova. Design of Formal Languages and Interfaces: "Formal" Does Not Mean "Unreadable". *Emerging Research and Trends in Interactivity and the Human-Computer Interface*. IGI Global, 2013.
24. M. Spichkova, H. Schmidt, and I. Peake. From Abstract Modelling to Remote Cyber-Physical Integration/Interoperability Testing. In *Improving Systems and Software Engineering Conference*. 2013.
25. M. Spichkova, X. Zhu, and D. Mou. Do We Really Need to Write Documentation for a System? In *International Conference on Model-Driven Engineering and Software Development*, 2013.
26. S. Dal Zilio, D. Lugiez, and C. Meyssonnier. A Logic You Can Count on. In *Symposium on Principles of programming languages*, ACM, 2004.

Towards a Base Model for UML and OCL Verification*

Frank Hilken, Philipp Niemann, Robert Wille, and Martin Gogolla

University of Bremen, Computer Science Department
D-28359 Bremen, Germany
{fhilken,pniemann,rwille,gogolla}@informatik.uni-bremen.de

Abstract. Modelling languages such as UML and OCL are more and more used in early stages of system design. These languages offer a huge set of constructs. As a consequence, existing verification engines only support a restricted subset of them. In this work, we propose an approach using model transformations to unify different description means within a so called base model. In the course of this transformation, complex language constructs are expressed with a small subset of so-called core elements. This simplification enables to interface with a wide range of verification engines with complementary strengths and weaknesses. Our aim is that, guided by a structural analysis of the base model, the developer can choose the most promising verification engine.

1 Introduction

In recent years Model-Driven Engineering (MDE), has become more and more important. In this context, the Unified Modelling Language (UML) and the Object Constraint Language (OCL) are de facto standards to describe systems and their behaviour. Identifying errors early in the design of such systems using validation and verification techniques is an important task, though finding the right verification approach is not trivial, since most approaches concentrate on only one UML diagram type and restrict the set of supported diagram types and language constructs.

In this paper, we propose the idea of a so-called *base model* that, using model transformations, combines the information of several UML diagram types into a single diagram and reduces the set of language constructs to a minimum. The language constructs are split into three categories: (1) core elements that are directly used in the base model; (2) transformed elements that are represented in the base model using only core elements, reducing the used amount of language constructs; and (3) unsupported elements that are excluded because of their low relevance or because of their infeasibility in the context of verification. The base model can be seen as a substantially reduced version of UML with a strong focus on compatibility to verification engines.

* This work was partially funded by the German Research Foundation (DFG) under grants GO 454/19-1 and WI 3401/5-1 as well as within the Reinhart Koselleck project DR 287/23-1.

On the basis of the base model, we perform a structural analysis separated into several categories in order to identify the most promising verification engine for the particular model under development. Once a verification engine is chosen, further model transformations can remove remaining incompatibilities in the same manner as from the source model to the base model. All previously mentioned transformations are performed on the UML and OCL layer in order to have a unified process for every verification engine.

The structure of the paper is as follows. In Sect. 2, the base model is discussed including the process flow, definitions and transformations. Section 3 describes the analysis of the base model and how a solver is chosen. Lastly, the transformations for the solving engine are shown in Sect. 4. In Sect. 5, we discuss related work before concluding the paper in Sect. 6.

2 A Base Model for UML and OCL Verification

The base model provides an interface between arbitrary UML/OCL model descriptions and validation and verification tools. The goal of the base model is to represent a unified basis for model descriptions combining various UML diagram types while retaining a maximum compatibility to the original model and to solvers.

Figure 1 shows the process flow when using such a base model. The source model description consists of various UML diagrams enhanced with OCL expressions in order to specify the system and its behaviour. All diagrams conjoined are transformed and combined into a base model. Using the base model, a structural analysis can provide hints for an appropriate solver selection. Once the solver is chosen, the base model is transformed into a solver-specific base model elimi-

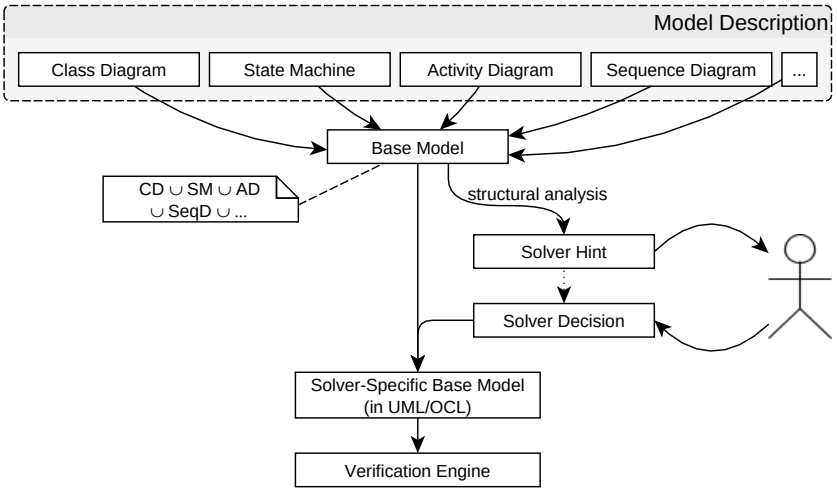


Fig. 1. Process flow employing the base model

nating modelling constructs not explicitly supported by the solver and replacing them with simpler representations. All transformations are automatic and do not require manual interaction unless the user wants to choose a specific solving engine. Finally, the solver specific base model serves as input for the verification engine.

We define the basis of the source model description to be a class diagram optionally enhanced with operations specifying model behaviour. Other diagram types, such as state machines, activity diagrams and sequence diagrams, can further define the system and its behaviour. The result of the model description transformation is a class diagram with optionally employed operations using pre- and postconditions to describe the system's behaviour. The information of the various diagram types is transformed into either class invariants or pre- and postconditions of the operations. Additionally, the base model only features a reduced set of UML and OCL features, the core elements. Elements that are not part of these are transformed into simpler representations using only core elements to increase compatibility to solving engines.

Example 1. For instance, an aggregation from the source model is transformed into an association plus an invariant, demonstrated in Fig. 2 with a simple mother child relationship.

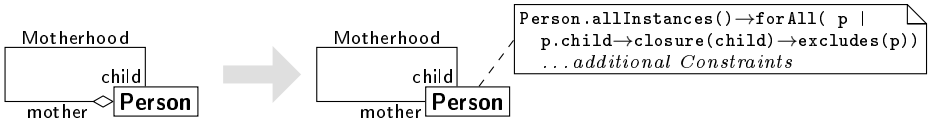


Fig. 2. Transformation of an aggregation into association plus constraint

A structural analysis on the base model is now able to determine the relevance of certain criteria and can give hints about which solving engines are most effective. This is described in more detail in Section 3. A developer can combine these hints and their knowledge about the system to choose the most appropriate solver. Alternatively, if the user interaction is not desired, the solving engine that the analysis has determined to be best can be chosen directly. In the case that the solving engine does not support all features present in the base model, these elements are further transformed into solver compatible elements, resulting in the solver specific base model. Most likely, these transformations are required to enable the usage of such verification engine anyway and therefore does not influence the scalability. Additionally, note that all transformations are performed on the UML and OCL layer and only have to be implemented once instead of individually for each solving engine.

Table 1. UML elements in the base model (*symbols:* ✓ core element; ∘ transformed element (using only core elements); × unsupported element)

Class features	Association features	Operation features
✓ Class	✓ Binary Association	✓ Operation (non query)
∘ Abstract Class	∘ N-ary Association	✓ Parameter
∘ Inheritance	∘ Aggregation	× Return Value
∘ Multiple Inheritance	∘ Composition	✓ Pre-/Postcondition
✓ Attribute	✓ Multiplicity	× Nested Operation Call
∘ Initial Value	∘ Association Class	∘ Query Operation
∘ Derived Value	∘ Qualified Association	✓ Parameter
✓ Enumeration	× Redefines, Subsets, Union	✓ Return Value
✓ Invariant		× Recursion

2.1 Elements of the Base Model

To create a unified model representing the input model description with a maximum compatibility to verification engines, the base model has a reduced feature set of UML and OCL elements. Table 1 shows the supported UML elements in the base model and marks those elements that are transformed into a more basic representation. The supported elements form a basis to represent most UML features either directly or using the available elements plus invariants and are supported by the majority of solving engines. Many transformations of complex model elements into the core elements are described in [7].

The essential elements of the base model are defined by the class and association features. The core elements, representing the "atoms" of the base model, are *enumerations*, *classes* with their *attributes* and *invariants*, and *binary associations* with their *multiplicities*. Most of these elements cannot be represented by simpler description means. Other structural features, such as *association classes* and *aggregations*, are transformed into representations using the core elements. Very specific features, like *redefines* and *subsets*, are not supported for now.

The operation features define the behaviour of the base model using *operations* with optional *parameters* and *pre-* and *postconditions*. *Return values* are not supported until *nested operation calls* are, which we can not handle currently. *Query operations* are integrated into the expression they are used in, which is also the reason why we do not support *recursion* for these.

As for OCL, we support a basic set of features to cover the majority of OCL expressions. The data types `Boolean` and `Integer` are fully supported. `Strings` are only represented by an ID, allowing for a comparison on an instance level, but not on a character level. This also excludes string operations like `concat`. We also support the OCL collection types `Set(T)`, `Bag(T)`, `Sequence(T)` and `OrderedSet(T)` with the essential collection operations, including but not limited to: constructors and manipulation operations `including` and `excluding`; membership tests `includes` and `excludes`; quantifiers `forAll` and `exists`; count operations `size`, `isEmpty` and `notEmpty`; filters `select` and `reject`; and `closure`.

The advantages of the transformation at this early stage are numerous. First and foremost, the solving engines do not require an encoding for the different UML elements. Instead, only the base elements have to be supported. The implementation of the transformation of the complex elements is only required once on the UML and OCL layer. Therefore, it is easier to provide support for more solving engines and they are easier interchangeable. There might be cases, in which a solving engine has a better encoding for a feature than the transformation in the base model, but we expect the transformed elements (marked with a $\circ$ symbol) to be affected least. Also, the assignment of elements to their respective category may change if case studies expose an advantage.

Secondly, the unification provides a solid baseline for the structural analysis to give optimal hints. The unification also helps to break down the model into the relevant categories like quantifiers and special OCL constructs, such as the `closure` expression.

Reasons, why certain elements are not part of the base model at all, are their incompatibility to be represented within the base elements and their high complexity which is not supported by many solving engines, e.g. nested operation calls. Also, elements depending on other unsupported elements are not part of the base model, e.g. return values for non query operations which are only useful if an operation invokes another operation to work with its return value. Note the difference between the transformation of UML and OCL features in the base model and the elimination of UML and OCL features in the solver specific base model, whose sole reason is the compatibility to the solving engine.

2.2 Incorporation of Various Diagram Types

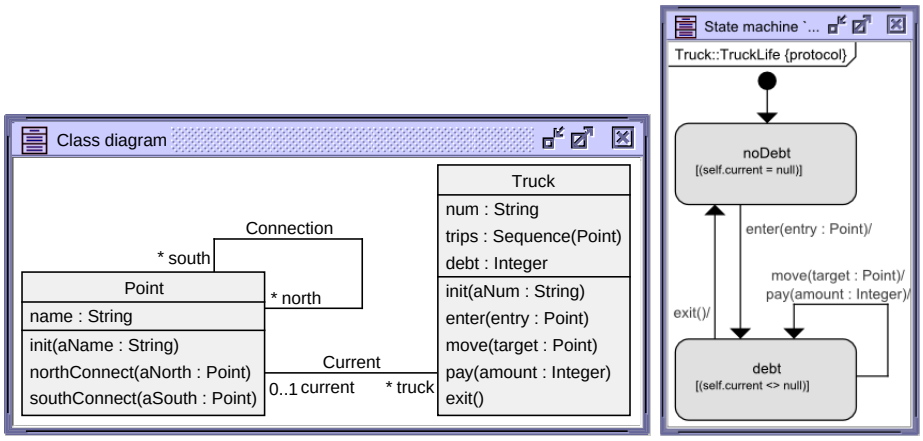
The various diagram types of UML provide different information about the model. A class diagram defines the data structure and operations of a system. State machines, for example, are able to further restrict operation invocations in addition to the operation preconditions. They can also define additional effects of operations as well as state invariants for object states. Activity diagrams provide operation behaviour in form of an implementation. Sequence diagrams again restrict the operation execution order on a different layer than state machines.

Most verification engines specialize on one of these diagram types or require a specific combination of them to be able to accept the system definition. We want to combine the information of the various diagram types into one base model using model transformations, accumulating all information. This way, the requirements for the verification engine are kept minimal and the selection of solvers increases.

In a first step, we plan to incorporate the information of protocol state machines – without events – into the base model, thereby combining class diagrams with operations and state machines.

Example 2. An example model specifying a Toll Collect[1] system is defined in Fig. 3. It consists of a class diagram (Fig. 3a) and a state machine (Fig. 3b).

[1] www.toll-collect.de/en/home.html



(a) Class diagram with operations (b) State machine

Fig. 3. Toll Collect example system

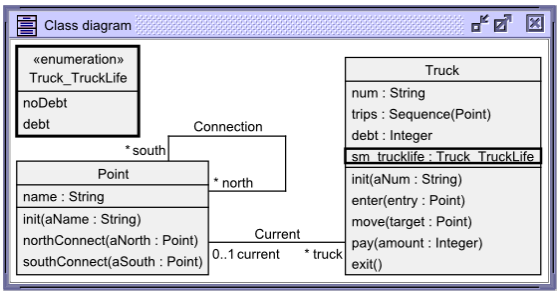
The system features a network of points on which trucks can enter and move around. For each visited point the debt of a truck increases. To leave the network, a truck has to pay its debt. The state machine ensures that the truck can only enter the network when it is not currently in it and only move around and exit the network when it is inside. The state invariants of the state machine further enforce these properties.

To transform the state machine into the class diagram, the states are represented as an enumeration. An additional attribute `sm_trucklife` is added to the corresponding class to save the current state. These changes are illustrated and highlighted in the class diagram in Fig. 4. Using the enumeration attribute, the state invariants can be represented as class invariants that only trigger if the object is in the required state. Constraints given by the transitions of the state machine, including guards and postconditions, are transformed into pre- and postconditions of the operation definitions in the class diagram, as shown on the right in Fig. 4.

3 Solver Selection

So far, various approaches for the verification of UML and OCL models have been presented [2,4,13]. The main idea of these approaches is to encode verification problems in a language that can be passed to a dedicated solving engine and interpret the results at the level of UML and OCL.

In this context, a large variety of languages and solving engines has been suggested. Approaches using *theorem provers* like Isabelle [2], reformulating the problem as a *Constraint Satisfaction Problem (CSP)* [4], using intermediate languages like *Alloy* or *Kodkod* [1,13] though finally resulting in an instance of




```
-- state invariant noDebt
context Truck inv:
  sm_trucklife = #noDebt
  implies current = null

-- op. Truck::exit()
pre: sm_trucklife = #debt
post: sm_trucklife =
    #noDebt
```

Fig. 4. State machine information integrated into class diagram

Boolean Satisfiability (SAT), or using a direct encoding in the more general language of *Satisfiability Modulo Theories (SMT)* [12] have been proposed.

All these solving engines rely on different abstractions and accordingly employ complementary solving schemes. Hence, it is important to choose an appropriate solver for a given UML and OCL description and verification problem. For instance, Kodkod and Alloy have been designed for problems of relational logic. They support set theory including transitive closure, and are, thus, especially suitable for OCL constraints expressing relations between model elements. In contrast, SMT offers theories for the efficient handling of bit-vectors (Integers) and corresponding arithmetic operations, while CSP particularly supports abstract data types (e.g., collections/lists). Beyond that, there are different approaches to cope with quantifiers (existential and universal) like *Quantified Boolean Formulas (QBF)* [6] or integration of quantifiers into bit-vector logic in the SMT-solver Z3 [5].

In order to benefit from these particular, complementary strengths, we suggest a structural analysis of the model, especially of the OCL expressions, with respect to collection types (e.g., `Set(T)` and `Sequence(T)`), arithmetic components (+, −, ∗, /), relational components (e.g., `size()` and `closure()`) and quantifiers (`forAll` and `exists`). This analysis shall provide hints which solver might be most adequate for the verification of the given base model. As adequateness can hardly be quantified in terms of absolute scores, the results shall be presented to the developer in an interactive procedure. First, the above criteria are listed together with their relevance for the given model (high, moderate, low or none) as determined by the structural analysis. These values may then be adjusted by the developer in order to incorporate her own rating. In a second step, a recommendation is given which solvers are most appropriate for the given problem based on how their strengths and weaknesses fit to the profile of relevance derived in the first step. This feedback shall also contain information about model elements that are not directly supported by the solvers (e.g. Reals, Strings, or certain OCL constructs) and will be ignored or transformed into simpler means. Assisted by this advice, the developer can finally decide on which solving paradigm might be most appropriate to use. Alternatively, the developer may also let this decision be made by the framework automatically.

4 Solver-Specific Base Model

Though many different solving technologies are available, basically all approaches do not cover the complete OCL language. They rather restrict themselves to a subset for which an encoding exists, i.e. a model transformation to the solver level. Consequently, before passing the base model to the chosen solver, unsupported model elements have to be transformed.

Example 3. Consider the collection type `Sequence(Point)`, as contained in the Toll Collect example model from earlier (c.f. Fig. 3a). If the chosen solver technology does support sequences (e.g. Kodkod, Alloy, or CSP) no transformation is necessary at this step and the base model is passed to the verification engine unchanged. However, if the chosen solver does not support sequences, these are to be transformed. This can be done by introducing two new classes `PointSequence` and `PointSequenceElement`, organized as a linked list with references to the beginning and the end of the sequence, shown in Fig. 5. The resulting solver-specific base model is then passed to the verification engine and finally to the solver.

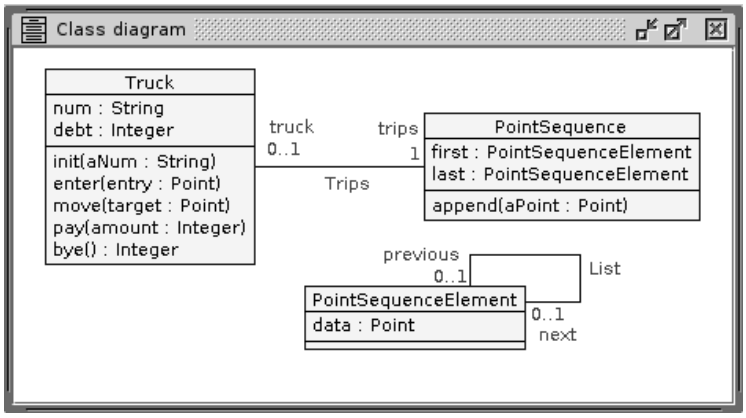


Fig. 5. Replacing OCL collection type *Sequence* by introducing a new UML class

UML and OCL are rich languages with many facets which makes it hard to (1) determine an encoding for each component (e.g. the encoding of collections in SMT [11]) and (2) requires a large effort to realize these encodings in (prototypic) implementations. Addressing this issue, the transformation to a solver-specific base model can also help to lower the threshold for incorporating new solving engines. More precisely, by providing a *whitelist* of supported UML and OCL components, the range of tractable model elements can be enlarged by OCL transformations (e.g. using `select` to express `reject` or using `COL→size()=0` for `COL→isEmpty()`) or by more substantial, structural modifications (e.g. transforming the collection type `Sequence(T)` to separate classes).

Though these transformations are performed automatically, the developer shall be able to access which kind of transformation is performed. This allows for a judgement whether they may interfere with the addressed verification task or whether they are applied to secondary components of the model only.

5 Related Work

There are several contributions that can be related to our present work. The fundamental idea of a base model within a generic verification methodology for system descriptions expressed in terms of UML and OCL has been presented in [14]. Verification of other description means than class diagrams has been addressed by many approaches, e.g. [8], while using OCL in order to express complex UML class diagram properties by simpler means, has been discussed in [7]. Validation and verification of such model transformations, e.g. using the ATLAS Transformation language (ATL) [3], is an active field of research. A comparison of such verification techniques has been presented in [9]. Also to the solver end, a similar, but more quantitative comparison between different solving paradigms has been conducted [10]. The results indicate a predominance of SMT, especially for large benchmarks. However, this comparison only considers a single class of models of the same type which are only varied in size, and a further comparison is needed for models of other types.

6 Conclusion and Future Work

In this work, we presented a model transformation of heterogeneous model descriptions to a unified base model, which enables to consider more comprehensive descriptions for verification. In the course of this transformation, a small set of core elements is used to express a large and rich set of UML and OCL constructs. In doing so and excluding several language constructs due to their minor relevance or general infeasibility with respect to verification, we expect to improve compatibility to verification engines without significantly restricting model support. We have exemplarily shown the incorporation of state machines into the base model by means of an example. Details of this transformation as well as the transformation of other diagram types are left for future work.

Beyond that, we have identified and suggested categories of modelling constructs that may affect the performance of verification engines, if these are applied to models that contain a considerable amount of those constructs. However, their actual impact on solving times and performance has not been examined thoroughly so far and remains open for further research.

Finally, by providing further transformations of the base model, we are able to obtain solver-specific base models employing only language constructs that are supported by the addressed solver. Instead of leaving the implementation of these transformations to the verification engines, they are performed transparently at a higher level of abstraction. This allows us to interface with a wider range of solvers, potentially at the price of a little overhead and loss of performance.

Overall, the base model framework provides us with a generic interface between heterogeneous model descriptions and verification engines.

References

1. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A challenging model transformation. In: Model Driven Engineering Languages and Systems, pp. 436–450. Springer (2007)
2. Brucker, A.D., Wolff, B.: The HOL-OCL book. Tech. Rep. 525, ETH Zurich (2006)
3. Büttner, F., Cabot, J., Gogolla, M.: On validation of ATL transformation rules by transformation models. In: Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation. p. 9. ACM (2011)
4. Cabot, J., Clarisó, R., Riera, D.: Verification of UML/OCL class diagrams using constraint programming. In: Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on. pp. 73–80. IEEE (2008)
5. De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008)
6. Giunchiglia, E., Narizzano, M., Tacchella, A.: Qube: A system for deciding quantified boolean formulas satisfiability. In: Automated Reasoning, pp. 364–369. Springer (2001)
7. Gogolla, M., Richters, M.: Expressing UML Class Diagrams Properties with OCL. In: Clark, T., Warmer, J. (eds.) Advances in Object Modelling with the OCL, pp. 86–115. Springer, Berlin, LNCS 2263 (2001)
8. Kaufmann, P., Kronegger, M., Pfandler, A., Seidl, M., Widl, M.: Global State Checker: Towards SAT-Based Reachability Analysis of Communicating State Machines. In: Boulanger, F., Famelis, M., Ratiu, D. (eds.) MoDeVVa@MoDELS. CEUR Workshop Proceedings, vol. 1069, pp. 31–40. CEUR-WS.org (2013)
9. Lano, K., Kolahdouz-Rahimi, S., Clark, T.: Comparing verification techniques for model transformations. In: Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation. pp. 23–28. ACM (2012)
10. Saadatpanah, P., Famelis, M., Gorzny, J., Robinson, N., Chechik, M., Salay, R.: Comparing the effectiveness of reasoning formalisms for partial models. In: Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation. pp. 41–46. ACM (2012)
11. Soeken, M., Wille, R., Drechsler, R.: Encoding OCL Data Types for SAT-Based Verification of UML/OCL Models. In: Gogolla, M., Wolff, B. (eds.) TAP. Lecture Notes in Computer Science, vol. 6706, pp. 152–170. Springer (2011)
12. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL Models Using Boolean Satisfiability. In: DATE. pp. 1341–1344. IEEE (2010)
13. Straeten, R.V.D., Puissant, J.P., Mens, T.: Assessing the Kodkod Model Finder for Resolving Model Inconsistencies. In: France, R.B., Küster, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA. Lecture Notes in Computer Science, vol. 6698, pp. 69–84. Springer (2011)
14. Wille, R., Gogolla, M., Soeken, M., Kuhlmann, M., Drechsler, R.: Towards a Generic Verification Methodology for System Models. In: Macii, E. (ed.) DATE. pp. 1193–1196. EDA Consortium San Jose, CA, USA / ACM DL (2013)

Bayesian Reasoning Over Models

Sebastian J. I. Herzig and Christiaan J. J. Paredis

Model-Based Systems Engineering Center (MBSEC),
G.W. Woodruff School of Mechanical Engineering,
Georgia Institute of Technology, Atlanta, GA, USA
sebastian.herzig@gatech.edu,chris.paredis@me.gatech.edu
http://www.mbsec.gatech.edu

Abstract. A crucial part of verifying and validating models is the identification of inconsistencies. Inconsistencies can exist whenever models overlap semantically. Such overlaps are predominant in model-driven engineering, where the use of multiple viewpoints leads to a variety of incomplete representations of one or more aspects of a system. While the commonly employed rule-based approaches to identifying inconsistencies can be effective, state of the art methods for inferring or determining semantic overlaps are not. Techniques relying on unification algorithms or a unifying ontology make strong assumptions, are error prone and can be costly to maintain. In this paper, an alternative approach based on Bayesian reasoning is proposed. We show how Bayesian inference combined with pattern matching can be used to infer likely semantic overlaps in models. The approach is illustrated and evaluated using the inference of semantic equivalences as an example of inferring one type of semantic overlap.

Keywords: inconsistency management, Bayesian reasoning, verification and validation, model composition

1 Introduction

When designing and developing complex systems, one common practice in model-driven engineering is for stakeholders to study the system from a variety of different viewpoints. Such viewpoints are defined by a number of factors, including the context, level of abstraction and concerns of interest [1]. Concerns of interest addressed from different viewpoints may overlap, leading to semantic relations and, hence, semantic overlaps among the corresponding views and models. Redundant definitions, for instance, imply semantic equivalence. Knowledge of such relations is required when verifying and validating models, particularly because violations of their intended semantics can lead to inconsistencies. While semantic overlaps can certainly be minimized by separating concerns as much as possible, a complete separation of concerns is rarely (if ever) possible.

Several methods for identifying semantic overlaps are proposed in the related literature. However, the associated cost, the strong assumptions made, and the fact that many of the techniques are error prone, renders them impractical for

most scenarios. For example, unification algorithms are typically based on name or predicate matching and can fail whenever homonyms or synonyms are encountered. Rule- or logical inference based approaches rely on rule antecedents to be matched completely. However, particularly when reasoning over incomplete information and knowledge, antecedents may not always (fully) match. This can lead to results and conclusions that are unintuitive to a human but logically correct [15]. In fact, there may be cases in which a human would have considered a partial match to provide sufficient evidence to suggest that the consequent of the rule should apply. Such behavior suggests that it is useful to account for uncertainty in automated reasoning processes. In this paper, we use this as a motivation for introducing a novel, Bayesian inference - based inexact reasoning approach for constructing probabilistic arguments about model-based information and knowledge, and apply the developed concepts to the problem of identifying semantic overlaps.

The remainder of the paper is organized as follows: section 2 briefly introduces the running example. Our conceptual approach is developed in section 3. A corresponding algorithm is introduced and evaluated in sections 3.3 and 3.4. Section 4 reviews similar approaches and compares them to our work. The paper closes with a short discussion and conclusions in section 5.

2 Running Example: Inferring Semantic Equivalence

To illustrate our conceptual approach, we apply it to the problem of inferring semantic overlaps. A semantic overlap implies the existence of a semantic relationship between two or more utterances of one or more languages. There are numerous kinds and types of semantic relationships. Some well-known relationships from object-oriented modeling are *meronomous* (part-whole, has a), *hyponymous* ("is a", i.e., type-of), *causal* and *instance-of* relations [12]. Particularly in Model-Driven Engineering (MDE), Model-Based Systems Engineering (MBSE), and generally multi-view and multi-paradigm software engineering, the additional category of *synonymy*-related relationships – which includes *semantic equivalence* – is of interest in identifying model correspondences and for the purpose of defining model transformations. We say that two or more expressions are semantically equivalent if they share a common semantic mapping (meaning).

In our running example, we assume that a number of UML models are given. As illustrated in figure 1a, some of these models contain classes with properties that have default values assigned. It is assumed that, across the different models, some of these properties may be semantically equivalent, but knowledge of their equivalence is not explicitly captured. The task is to find those pairs of properties that are likely to be semantically equivalent. Because models describing engineering systems (software and physical systems) are often heterogeneous in nature, and a great number of very different formalisms is typically employed, we also assume the existence of a (bi-directional) mapping to some common representational formalism for all models. For purposes of illustration and mathematical elegance, and to build on previous work, directed, attributed

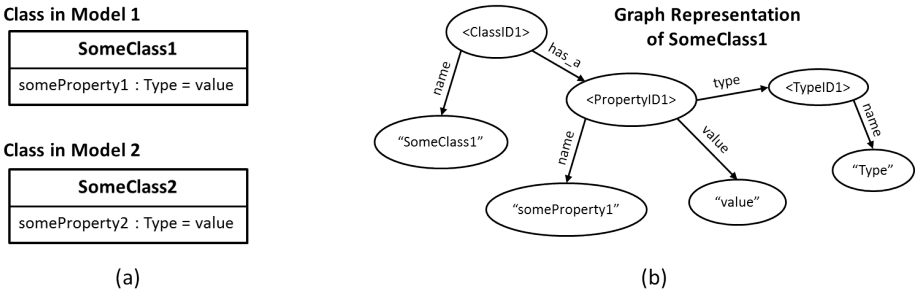


Fig. 1. Running example: (a) UML classes with properties that have default values assigned; (b) automatically generated graph representation of *SomeClass1* (extract).

(and typed) multi-graphs are used as a common representational formalism [9]. This is illustrated in figure 1b.

3 Bayesian Reasoning in Models Represented by Graphs

Bayesian reasoning is often considered similar to human reasoning [13]. Illustrative of this are situations in which humans are asked to classify objects. Without any information about the object (other than its existence), a human's belief about the class that the object belongs to is his or her *subjective belief*, which is typically formed on the basis of past experience. Once exposed to the object, a human tends to look for certain *features*, each of which provides further clues towards which class the object is likely to belong to. These features can be identified by *observing* the object – a process that can be interpreted as the collection of additional information (or *evidence*) that can be used to update a belief to form a *posterior belief*. We argue that the same principles can be applied to reasoning over model-based information and knowledge.

3.1 Bayesian Inference & Belief Networks

In Bayesian probability theory, beliefs are updated with new information by applying *Bayes' theorem* [2]. The belief to be updated is then also referred to as the *prior* belief. Assuming that A, B and C are observed events, and given a prior belief about A, the posterior belief about A can be updated with observations B and C using Bayes' theorem:

$$P(A \mid B, C) = \frac{P(A, B, C)}{P(B, C)} = \frac{P(A)\ P(B, C \mid A)}{P(B, C)}\ . \tag{1}$$

Determining the joint probabilities required to compute equation 1 is non-trivial. In part, this is due to values of joint probability distributions rarely being readily accessible. Also, when determining the joint probabilities by means of

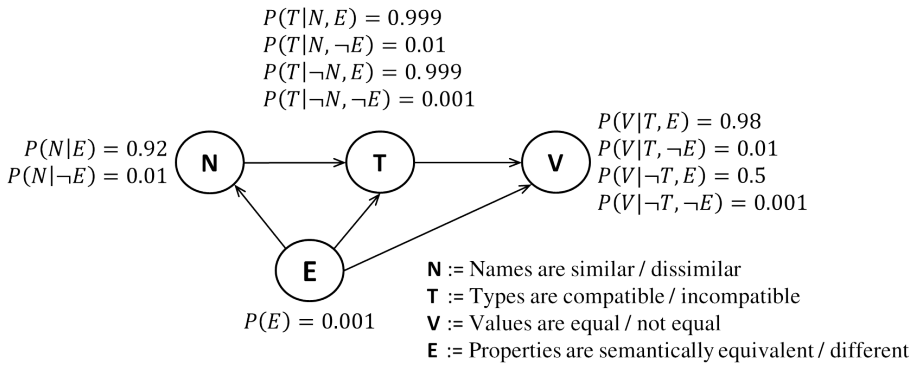


Fig. 2. Bayesian belief network for the running example – nodes denote random variables, arrows denote influences.

factorization, the number of terms required is (in general) very large, even for a small number of random variables [11].

Bayesian belief networks address both the problem of representing the joint probability distribution over a set of random variables and performing inference with these. Formally, a Bayesian belief network is a tuple $(\boldsymbol{G}, \mathcal{P})$, where $\mathcal{P}$ is a joint probability distribution over a set of random variables $\boldsymbol{\mathcal{V}}$ and $\boldsymbol{G}$ is a directed acyclic graph whose nodes are random variables in $\boldsymbol{\mathcal{V}}$. Directed edges are used to indicate *influence* or *causal* relationships, which express local dependence among random variables [11].

In addition to the DAG property, $(\boldsymbol{G}, \mathcal{P})$ must also satisfy the *Markov condition*. The Markov condition states that each variable is (locally) conditionally independent of its non-descendants given its parent variables. This condition, along with information about the conditional dependence significantly reduces the number of terms required to fully define the joint probability distribution $\mathcal{P}$ represented by a Bayesian belief network. Therefore, to determine the joint probability through simple enumeration, the product of the conditional probabilities of all random variables $\mathbf{X}_i \in \boldsymbol{\mathcal{V}}$ given values of their parents $pa\,(\mathbf{X}_i)$ (whenever these conditional distributions exist) must be determined [11] (see equation 2):

$$\mathcal{P} = P(\mathbf{X}_1 = x_1, \mathbf{X}_2 = x_2, ...) = \prod_i P(\mathbf{X}_i = x_i \mid pa(\mathbf{X}_i)) \; . \tag{2}$$

This reduces the set of unknowns to only the conditional distributions of the random variables in $\boldsymbol{\mathcal{V}}$ given values of their parents in the Bayesian network. These distributions are known as the *parameters* of a Bayesian network.

Figure 2 illustrates both the structure and the parameters of the Bayesian network used for the running example. Note the use of the short hand notation $P(\mathbf{X}_i = 0) = P(X_i)$ and $P(\mathbf{X}_i = 1) = P(\neg X_i)$ for binary random variables $\mathbf{X}_i$. The network is assumed to be constructed by a human and encodes the assumption that knowing whether or not a particular pair of properties has similar (or

dissimilar) names (**N**), compatible (or incompatible) types (**T**), and equal (or unequal) values (**V**) is influenced by whether or not the pair of properties is semantically equivalent (or different) (**E**). In addition, it is assumed that whether or not names are similar influences the probability of types being compatible which, in turn, influences the probability of values being equal. The parameter values shown reflect the subjective beliefs of the same human. For example, at the time of specifying the network parameters, the human believes that the probability of any pair of properties being semantically equivalent is 0.1%.

3.2 Using Pattern Matching to Measure Random Variables

Using the information provided in the Bayesian network illustrated in figure 2, as well as equations 1 and 2, a number of interesting *diagnostic* inferences can be performed. For instance, consider an experiment where the random outcome $\omega \in \Omega$ is a pair of properties from the space of all pairs of properties Ω. Say one can determine (by observing the object) that the pair of properties (ω) has similar names, unequal values and compatible types. Furthermore, say that, due to a lack of available information and knowledge, we cannot be certain about the semantic equivalence of the two properties. Taking all of this new information into account, the probability of semantic equivalence for this particular pair of properties (which, without the observations is only the belief of any pair of properties being semantically equivalent: i.e., $P(E)$) can be updated. Mathematically, this equates to determining $P(E \mid N, \neg V, T)$. Note that $P(E \mid N, \neg V, T)$ is a meaningful statement about the probability of semantic equivalence of *any* pair of properties for which it can be determined with certainty that their names are similar, types compatible and values not equal.

By the earlier assumption that all models are represented by a graph, the definition of the UML class properties considered in the running example must also be represented by a graph (at least at some level of abstraction – see figure 1b). Determining whether or not any two properties represented by a graph fulfill a certain condition (e.g., such as both properties having similar names) can be done computationally by means of graph pattern matching. Therefore, we argue that the process of collecting more information about, e.g., a particular pair of UML class properties can be mapped to querying a graph pattern.

To illustrate this result more formally, let $(\Omega, \mathcal{F}, P)$ represent the probability space over which all random variables $\mathbf{X}_i \in \boldsymbol{\mathcal{V}}$ in the Bayesian network are defined. We define the sample space Ω as the set of all pairs of property definitions $N_{\mathcal{G},prop}$ in the graph $\mathcal{G}$ representing models: $\Omega = N_{\mathcal{G},prop} \times N_{\mathcal{G},prop}$. Furthermore, we define the σ-algebra as $\mathcal{F} = 2^{\Omega}$ (where 2^{Ω} denotes the power set). By definition, a random variable is a mapping $\mathbf{X}_i : \Omega \to E$ from the sample space to some measurable space E. Therefore, an $e \in E$ must be measurable for any $\omega \in \Omega$. By definition of the measurable preimage $\mathbf{X}_i^{-1}(e) \in \mathcal{F}$, an $e \in E$ is measured whenever an event $f \in \mathcal{F}$ is observed. To fully define the mapping, it is sufficient to determine which pairs of properties are elements of the preimages of a random variable. Per our definition of the random variable **N**, the preimage of $\mathbf{N}^{-1}(0)$ is the set of all pairs of properties with similar names, i.e., all $\omega \in \Omega$

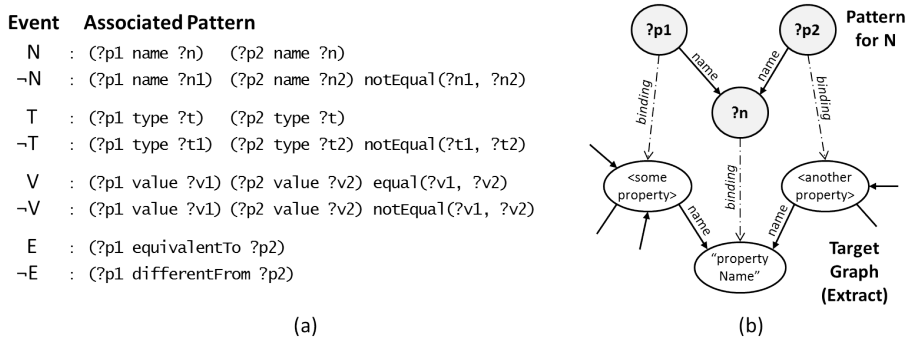


Fig. 3. (a) Patterns used in running example; (b) graph representation of the pattern associated with event N with example node variable bindings for a single match.

which have similar names. We argue that determining the pairs of properties (i.e., the ω_is) for which this is the case, is computationally possible by encoding the necessary knowledge in an appropriate pattern.

Figure 3 illustrates the patterns used for the running example in a datalog-like syntax. Variables are unique by name and are indicated by a `?` as prefix. *Graph triples* – i.e., two nodes (a *subject* and an *object*) connected by a directed edge (a *predicate*) – are separated by brackets and are written in the form `(subject predicate object)`. Note that `notEqual(x, y)` and `equal(x, y)` are functors that perform semantic equality checks on their arguments (for instance, 1 and 1.0 are considered semantically equal).

Note carefully that, per the definition of the probability space, every property can and, in a state of perfect information and knowledge, must have a name, type and value. This means that all pairs of properties must be a part of one of the preimages $\mathbf{X}_i^{-1}(e)$. In demonstrating our approach, only binary random variables were used. However, we recognize that multi-valued random variables may be more appropriate in other cases. Also note that, in a state of incomplete or inconsistent information, only a subset of the members of each of the preimages of the random variables can be determined using only the knowledge encoded in a pattern. Additional resources need to be committed to determine set membership for the other pairs of properties. Committing additional resources may require human intervention and the adding of additional information to the models.

3.3 Algorithm

By exploiting the structure of the Bayesian network, inference of probability distributions can be performed quite efficiently, e.g., using the *junction tree* algorithm [11]. Less trivial is the process of collecting information about a particular pair of properties – i.e., computationally determining which information and knowledge should be taken into account when determining the probability of semantic equivalence for a particular pair of properties. For instance, for a pair of

Algorithm 1: Infer propositions and associated probability distributions given a set of changes to a graph and a Bayesian network.

```
Algorithm doInference(Graph G, Triples T, BayesianNetwork B)
    for t ∈ T do
        InfContext ⟵ observe(G, t, events(B), true) ;
        for observations ∈ InfContext do
            ObservedRVs ⟵ observations.getRandomVariables() ;
            for rv ∈ (B.getRandomVariables() \ ObservedRVs) do
                D ⟵ D ∪ B.inferDistribution(rv, ObservedRVs) ;
    return D

Procedure observe(Graph G, Triple t, Events E, Boolean expand)
    for e ∈ E do
        p ⟵ pattern(e) ;
        T_L ⟵ ∅ ;
        if tripleMatchesPartOfPattern(p, t) then
            while (m ⟵ G.nextMatch(t, p)) ≠ ∅ do
                B_s ⟵ m.getBindingsToSharedVariables() ;
                Obs[B_s] ⟵ Obs[B_s] ∪ (e, variableBindings(m)) ;
                if expand then
                    for b ∈ B_s do
                        T_L ⟵ T_L ∪ G.findTriplesAbout(b) ;
        for t_l ∈ T_L do
            Obs ⟵ Obs ∪ observe(G, t_l, E \ e, false) ;
    return Obs
```

properties that has no values assigned, but similar types and names, $P(E \mid N, T)$ constitutes a meaningful, and, using the Bayesian network, inferable probability of semantic equivalence, since it takes all of computationally determinable information into account – that is, all of information that can be extracted from the graph solely using graph pattern matching. To determine the probability of semantic equivalence for each individual pair of properties, a naive algorithm would have to iterate over all pairs of properties and, for each pair, a number of graph searches would need to be performed to determine matches to all patterns associated with the random variables. In addition, a potentially large number of inferences in the Bayesian network need to be performed.

Given the complexity of these operations, we propose an incremental algorithm (see algorithm 1) which considers only the changes made to an input graph. The changes are provided in the form of a set of graph triples. The incremental behavior of the algorithm is valid for as long as the structure of the Bayesian network does not change (note that a change in the parameters would only require a re-computation of the posterior beliefs). Verbally, algorithm 1 attempts to measure all random variables within the context of a single triple by matching the

associated patterns, followed by performing inference in the Bayesian network. This procedure is called iteratively over the set of added triples: first, the current triple is compared to all patterns. If the triple can be matched against any part of a pattern, a full pattern match in the graph is performed. For each match to the pattern, the value of the random variable and a *context* defined by the bindings to the common, shared pattern variables (in the running example: `?p1` and `?p2`) is stored in a map. To find matches to the other patterns within one variable binding context, a list of triples directly related to the nodes and edges bound to the variables shared across patterns is compiled (i.e., triples with one of the bound elements as a subject, predicate or object). The pattern matching procedure is then repeated over this list.

3.4 Proof-of-Concept Implementation & Algorithm Evaluation

In previous work, we have developed a model-based reasoning framework called ConSystent [9]. ConSystent uses the Resource Description Framework (RDF) as an underlying formalism for representing models by graphs. RDF representations of models are automatically generated. This generated RDF data is collected and stored in a central RDF store (Apache Fuseki).

For the purpose of demonstrating the technical feasibility of our approach, and to test our algorithm, we have extended this existing infrastructure with two additional components: firstly a simple Bayesian network library supporting inference with discrete random variables, and secondly a custom reasoning engine which implements Apache Jena's *Reasoner* interface. The expressiveness for defining patterns using the Apache Jena datalog-like rule language is preserved by internally rewriting the patterns used for measuring random variables as rules with empty rule headers.

A preliminary evaluation of the algorithm and its implementation was performed using three sets of models. In each scenario, different quantities, kinds (UML, Simulink) and sizes of models were used. Samples of the inference results were drawn at random and inspected manually. Two important reflections have been made: firstly, common inferences (such as pairs of properties, for which the only observation is type inequality) can lead to a very large number of inferences, even for small models. This indicates that patterns need to not only be designed carefully, but heuristics may need to be employed to further define which inferences are considered *valuable*. Such heuristics can then form a basis for a classifier that decides which inferences to present to a modeler for further consideration. Secondly, due to the algorithm iterating over the set of all triples, some inferences are performed multiple times. However, this is not considered an issue – rather, it is a likely indicator of a non-optimality of the algorithm.

4 Related Work

In the related literature, most approaches to reasoning over (model-based) information and knowledge are based on logical inference. Inference of semantic overlaps typically makes use of unification algorithms, which exploit representation

conventions. This includes predicate matching, of which the work by Finkelstein et al. is an example [6]. Additionally, Triple Graph Grammars (TGG) and, more generally, correspondence models have been used for similar purposes [7]. The approaches are similar in that syntactic matching is performed. Rules are used to define correspondences and transformations, which are typically based on structural semantics. In some instances, models are enhanced with stereotypes [14] or elements from a common, shared ontology [8]. However, all of these methods make a number of strong assumptions: for example, in name- or predicate-based approaches, spelling mistakes or the use of synonyms can produce false negatives. False positives may be produced as a result of homonyms. Common, shared ontologies can be criticized based on the argument that necessitating tagging of models with elements of the ontology is highly labor intensive and agreement of the ontology among stakeholders is difficult to achieve [15]. Secondly, unifying ontologies can quickly grow unmanageably large at which point they become expensive to maintain.

Approaches to inferring semantic overlaps that are based on similarity analysis can be considered complementary to our work. In [3, 5] probabilistic approaches to mediating database schemas are introduced. Probabilistic inference has also been investigated in semantic web applications. For instance, in [10], an extension to RDF was proposed to support uncertain inferences by associating probabilities with both implications and statements. PR-OWL, a proposed probabilistic extension to the *Web Ontology Language* (OWL) to define Bayesian networks, is introduced in [4]. The disadvantage to most of these approaches is that they are either not Bayesian – which, by definition, does not provide a suitable basis for admissible decision rules [2] – or are incomplete, or provide no working implementation.

5 Discussion & Conclusions

In this paper, the use of Bayesian inference in combination with pattern matching is demonstrated and applied to the problem of inferring (likely) semantically equivalences. Identifying semantic overlaps – and generally reasoning over models – is an essential part of identifying inconsistencies and, hence, indispensable to verification and validation of models.

Logical inference can fail in scenarios where incomplete, underspecified and inconsistent views of models are consolidated. Bayesian inference, on the other hand, can always draw useful conclusions. Combining Bayesian inference and pattern matching as described in this paper can be viewed as an extension to the more commonly applied approach of using implications (or, generally, rules) to perform logical inference, and model- and graph-transformations: any outcome with probability 1 or 0 can be said to have been logically entailed by the evidence considered. For these reasons, applications such as *spam filtering* employ a similar combination of Bayesian inference and pattern matching to improve the effectiveness of the reasoning task at hand.

To the best of knowledge of the authors, the combination of Bayesian inference and graph pattern matching has, to the date of writing this paper, not been used within the context of reasoning about properties of engineering models. However, we strongly believe that such an approach is promising, particularly for large-scale model-driven development applications. This is supported by the fact that Bayesian inference allows for rational assessment of important properties, e.g., related to the state of consistency and validity, of incomplete, underspecified and inconsistent models.

Acknowledgments. This work was supported by Boeing Research & Technology. The authors would like to thank Michael Christian (The Boeing Company), Dr. Vinod Cheriyan (MBSEC) and the anonymous reviewers for their feedback.

References

1. ISO / IEC 42010 Systems and Software Eng.: Architectural Description (2007)
2. Berger, J.O.: Statistical Decision Theory and Bayesian Analysis. Springer (1985)
3. Berlin, J., Motro, A.: Database Schema Matching using Machine Learning with Feature Selection. In: Advanced information systems engineering. pp. 452–466. Springer (2002)
4. Costa, P.C., Laskey, K.B.: PR-OWL: A Framework for Probabilistic Ontologies. Frontiers in Artificial Intelligence and Applications 150, 237 (2006)
5. Doan, A., Domingos, P., Halevy, A.Y.: Reconciling Schemas of Disparate Data Sources: A Machine-learning Approach. SIGMOD Rec. 30(2), 509–520 (May 2001), http://doi.acm.org/10.1145/376284.375731
6. Finkelstein, A.C., Gabbay, D., Hunter, A., Kramer, J., Nuseibeh, B.: Inconsistency Handling in Multiperspective Specifications. IEEE Transactions on Software Engineering 20(8) (1994)
7. Giese, H., Wagner, R.: From Model Transformation to Incremental Bidirectional Model Synchronization. Software & Systems Modeling 8(1) (2009)
8. Hehenberger, P., Egyed, A., Zeman, K.: Consistency Checking of Mechatronic Design Models. In: Proceedings of IDETC/CIE (2010)
9. Herzig, S.J., Qamar, A., Paredis, C.J.: An Approach to Identifying Inconsistencies in Model-based Systems Engineering. Procedia Computer Science (2014)
10. Meiser, T., Dylla, M., Theobald, M.: Interactive Reasoning in Uncertain RDF Knowledge Bases. In: Proceedings of the 20th ACM international conference on Information and knowledge management. pp. 2557–2560. ACM (2011)
11. Neapolitan, R.E.: Probabilistic Reasoning in Expert Systems: Theory and Algorithms. CreateSpace Independent Publishing Platform (2012)
12. OMG: OMG Unified Modeling Language (OMG UML). Tech. rep. (2011), http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF
13. Pearl, J.: Fusion, Propagation, and Structuring in Belief Networks. Artificial intelligence 29(3), 241–288 (1986)
14. Shah, A.A., Kerzhner, A.A., Schaefer, D., Paredis, C.J.: Multi-View Modeling to Support Embedded Systems Engineering in SysML. In: Graph Transformations and Model-Driven Engineering, pp. 580–601. Springer (2010)
15. Spanoudakis, G., Zisman, A.: Inconsistency Management in Software Engineering: Survey and Open Research Issues. Handbook of Software Engineering and Knowledge Engineering 1 (2001)

Colored Petri Net-based Modeling and Formal Analysis of Component-based Applications

Pranav Srinivas Kumar, Abhishek Dubey and Gabor Karsai

Institute for Software Integrated Systems
Department of Electrical Engineering and Computer Science
Vanderbilt University, Nashville, TN 37235, USA
{pkumar, dabhishe, gabor}@isis.vanderbilt.edu

Abstract. Distributed Real-Time Embedded (DRE) Systems that address safety and mission-critical system requirements are applied in a variety of domains today. Complex, integrated systems like managed satellite clusters expose heterogeneous concerns such as strict timing requirements, complexity in system integration, deployment, and repair; and resilience to faults. Integrating appropriate modeling and analysis techniques into the design of such systems helps ensure predictable, dependable and safe operation upon deployment. This paper describes how we can model and analyze applications for these systems in order to verify system properties such as lack of deadline violations. Our approach is based on (1) formalizing the component operation scheduling using Colored Petri nets (CPN), (2) modeling the abstract temporal behavior of application components, and (3) integrating the business logic and the component operation scheduling models into a concrete CPN, which is then analyzed. This model-driven approach enables a verification-driven workflow wherein the application model can be refined and restructured before actual code development.

1 Introduction

Safety and mission-critical DRE systems are used in a variety of domains such as avionics, locomotive control, industrial and medical automation. Given the increasing role of software in such systems, growing both in size and complexity, utilizing predictable and dependable software is critical for system safety. To mitigate this complexity, model-driven, component-based software development has become an accepted practice. Applications are built by assembling together small, tested component building blocks that implement a set of services. Models describe what these component blocks are, what interfaces they have, how they are built, how they interact and how they are deployed to realize the domain-specific application.

Complex, managed systems, e.g. a fractionated spacecraft following a mission timeline and hosting distributed software applications expose heterogeneous concerns such as strict timing requirements, complexity in deployment, repair and integration; and resilience to faults. High-security and time-critical software applications hosted on such platforms run concurrently with all of the system-level mission management and failure recovery tasks that are periodically undertaken on the distributed nodes. Once deployed, it is often difficult to obtain low-level

access to such remote systems for run-time debugging and evaluation. These types of systems therefore demand advanced design-time modeling and analysis methods to detect possible anomalies in system behavior, such as unacceptable response time, before deployment.

Our team has designed and prototyped a comprehensive information architecture called **D**istributed **RE**al-time **M**anaged **S**ystem (DREMS) [1, 2] that addresses requirements for rapid component-based application development. In prior work, we have described the design-time modeling capability [3], and the component model used to build and execute applications [4]. The formal modeling and analysis method presented in this paper focuses on applications that rely on this foundational architecture.

The principle behind this design-time analysis here is to map the structural and behavioral specifications of the system under analysis into a formal domain for which analysis tools exist. Using an appropriate model-based abstraction, the mapping from one domain to another remains valid under successive refinements in system development, including code generation. Application developers use domain-specific modeling languages to model the component assembly, component interactions, component execution code, operation sequencing, and associated temporal properties such as estimated execution times, deadlines etc. Using such application-specific parameters in the *design* model, a Colored Petri net-based (CPN) [5] *analysis* model is generated. The analysis must ensure that, under the assumptions made about the components and the component architecture, the behavior of the system remains within the safe operational region. The results of this analysis will enable system refinement and re-design if required, before actual code development.

The remainder of this paper is organized as follows. Section 2 presents existing research relating to this paper; Section 3 provides a brief background on the DREMS Infrastructure and on the CPN formalism; Section 4 discusses the problem statement that is evaluated; Section 5 describes how this architecture is abstracted and modeled using CPN; Section 6 investigates the utility and scalability of state space analysis; Section 7 briefly describes how the analysis model is generated; Sections 8 and 9 present future extensions to the proposed approach and concluding remarks respectively.

2 Related Research

In recent years, much of the proliferating work in the development of mission-critical distributed real-time systems addresses the need for Safety and Verification driven Engineering. Structural properties of the system are established using domain-specific modeling tools. Design models are transformed into relevant analysis models to study possible behaviors of the system and identify anomalies. When analyzing timing behavior, typically several exaggerated assumptions such as upper bounds on task execution times, service rates, maximum resource consumption etc are made. The results of system analysis using these assumptions are equally pessimistic. However, real-time systems with high criticality necessitate such assumptions to avoid the consequences of poor design.

Predictability of system behavior is achieved by obtaining upper bounds on the system properties.

Petri nets and their extensions have proven to be a powerful formalism for modeling and analyzing concurrent systems. System designs represented using a domain-specific modeling languages are often translated into Petri nets for formal analysis. High-level formalisms such as AADL models have been translated into Symmetric Nets for qualitative analysis [6] and Timed Petri nets [7] to check for real-time properties such as deadline misses, buffer overflows etc. Similar to [7], our CPN-based analysis also makes use of observer places [8] that monitor the system behavior and look for real-time property violations and prompt completion of operations. However, [7] only considers periodic threads in systems that are not preemptive. Our analysis covers a broader range of thread interaction patterns geared towards component-based applications operating on a hierarchical scheduling scheme requiring higher-level modeling concepts to capture component interaction in a distributed setup.

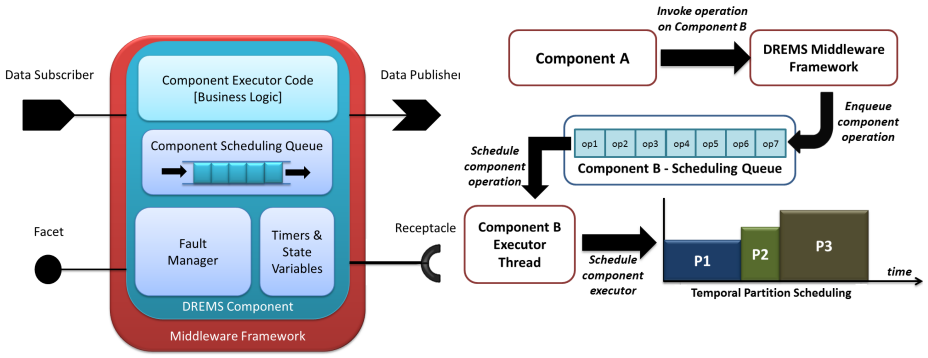
In the context of component-based systems, for complete real-time analysis, significant information must be obtained about the component assembly, the interaction patterns and the temporal behavior of components. The real-time model of the system is composed of real-time models of its constituent parts, each with its own temporal behavior. Using abstract model descriptors, [9] describes a real-time model for component-based systems, including semantic and quantitative meta-data about component real-time behavior. Using the MAST transactional modeling methodology [10] and analysis tools in the MAST environment, schedulability checks and priority assignment automation are performed. Note here that for every real-time application, a separate and independent real-time analysis model is generated for each mode of operation and analyzed separately.

For classes of component-based systems whose component assembly and application structure change dynamically over time, design-time verification is observed to be insufficient. Incremental re-verification strategies [11] have been applied to dynamic systems to augment traditional compositional verification by identifying the minimal set of components that require re-verification after dynamic changes. Since our approach considers design-time deployment plans that are static, our analysis does not consider dynamic changes to component assembly at run-time, but it will be subject of future work.

3 Background

DREMS Components: Design and implementation of component-based software applications rests on the principle of assembly: *Complex systems are built by composing re-useable interacting components.* Components contain functional, business-logic code that implements operations on state variables. Ports facilitate interactions between communicating components. A component-level message queue, with associated infrastructure code, controls the scheduling of operations of the individual components. Figure 1a shows the basic DREMS component.

Each DREMS component supports four basic types of ports for interaction with other collaborating components: Facets, Receptacles, Publishers and Subscribers. A component's **facet** is a unique interface that can be invoked either



(a) DREMS Component (b) Component Operation Scheduling

Fig. 1: DREMS Infrastructure

synchronously via remote method invocation (RMI) or asynchronously via asynchronous method invocation (AMI). A component's **receptacle** specifies an interface required by the component in order to function correctly. Using its receptacle, a component can invoke operations on other components using either RMI or AMI. A **publisher** port is a single point of data emission and a **subscriber** port is a single point of data consumption. Communication between publishers and subscribers is contingent on the compatibility of their associated topics (i.e. data types). More details on this component model can be found in [4].

Component Operation Scheduling: An operation is an abstraction for the different tasks undertaken by a component. These tasks are implemented by the component's executor code written by the developer. As shown in Figure 1b, in order to service interactions with the underlying framework and with other components, every component is associated with a message queue. This queue holds instances of operations ('messages') that are ready for execution and need to be serviced by the component. These operations service either interaction requests (seen on communication ports) or service requests (from the underlying framework). An example for the latter is the use of component timers that can periodically (or sporadically) activate an operation. Each operation is characterized by a priority and a deadline. The deadline here is the maximum acceptable time between the release of a component operation and the completion of that operation, measured starting from when the operation is enqueued onto the component's message queue. To facilitate component behavior that is free of deadlocks and race conditions, the component's execution is handled by a single thread. This single-threaded execution helps avoid synchronization primitives such as internal lock variables that lead to tenuous code development.

The DREMS OS scheduler enforces an ARINC-653 [12] style temporal and spatial partition scheme in order to schedule components grouped into processes. Temporal partitions, as shown in Figure 1b, are periodic fixed intervals of the processor time. Note that there are two levels of scheduling in DREMS: (1) Each component operation in the component-level is scheduled using a component-level scheduler, and (2) each component executor thread, on the system-level, is

scheduled by the OS in one of the temporal partitions, granting a slice of the CPU's time to those threads.

3.1 Colored Petri Nets

Petri nets [13] are a graphical modeling tool used for describing and analyzing a wide range of systems. A Petri net is a five-tuple $(P, T, A, W, M0)$ where P is a finite set of places, T is a finite set of transitions, A is a finite set of arcs between places and transitions, W is a function assigning weights to arcs, and M0 is the initial marking of the net. Places hold a discrete number of markings called tokens. A transition can legally fire when all of its input places have necessary number of tokens, and when fires it produces tokens for its output places.

With Colored Petri nets (CPN) [5], tokens carry values of specific data types called colors. Transitions in CPN are enabled for firing only when valid colored tokens are present in all of the typed input places, and valid arc bindings are realized to produce the necessary colored tokens on the output places. The firing of transitions in CPN can check for and modify the data values of these colored tokens. Furthermore, large and complex models can be constructed by composing smaller sub-models as CPN allows for hierarchical description. This extended paradigm can more easily model and analyze systems with typed parameters.

4 Problem Statement

Consider a set of mixed-criticality component-based applications that are distributed and deployed across a cluster of embedded computing nodes. Each component has a set of interfaces that it exposes to other components and to the underlying framework. Once deployed, each component works by executing operations placed on its component message queue. Each component is associated with a single executor thread that handles these operation requests. These executor threads are scheduled in conjunction with a known set of highly critical system threads and low priority best-effort threads. Furthermore, the application threads are also subject to a temporally partitioned scheduling scheme. System assumptions include (1) knowledge of the sequence of computational steps of known duration that are executed inside each component operation, (2) knowledge of the worst-case estimated time taken by each computational step, and (3) the estimated worst-case time taken to initiate a remote function call and to process the response, accounting for network-level delays. Using this knowledge about the system, the problem here is to ensure that the temporal behavior of all the application components lies within the bounds laid out by the system specifications. Ideally, this is achieved by verifying such system properties as lack of deadline violations for component operations. For scenarios where the system design isn't complete, e.g. application thread priorities are unknown, the paper describes the utility of an approach to identifying the subset of system behaviors that satisfy timing requirements and provide useful information to designers, e.g. partial thread execution orders.

5 Colored Petri net-based Modeling

This section briefly describes how CPN can be used to build an extensible, scalable analysis model for component-based software applications. To edit, simulate and analyze this model, we use the CPN Tools [14] tool suite.

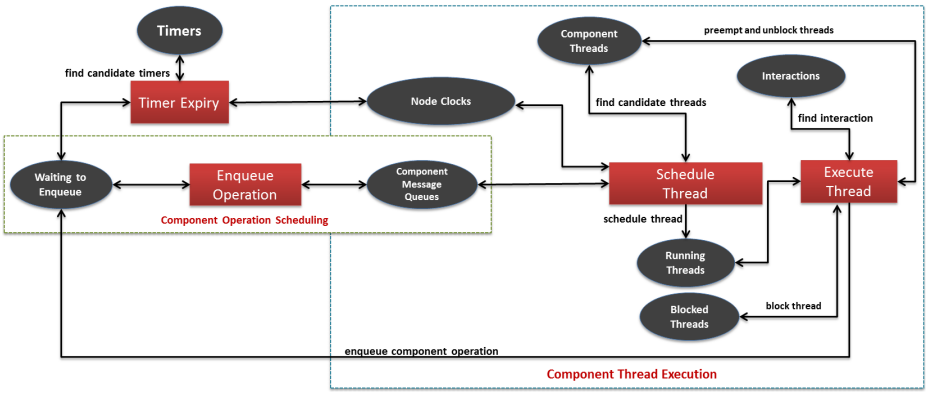


Fig. 2: Hierarchical CPN Analysis Model

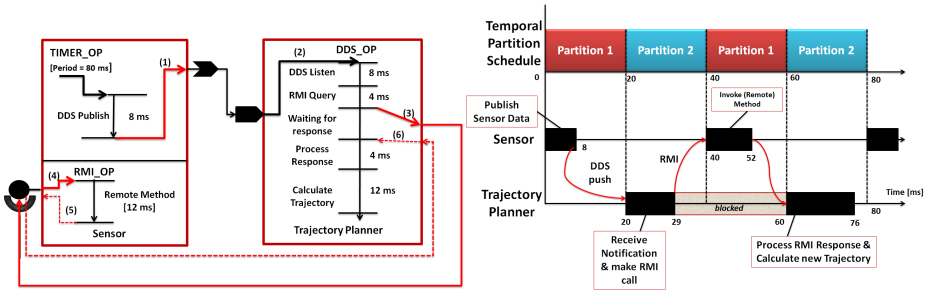
The CPN model captures the behavioral semantics of our component model described in [4], using knowledge of several factors that resolve the deployment of the component-based application. These factors include the following system properties: (1) configuration of temporal partition scheduling on each node of the distributed system, (2) location of each component being deployed (which temporal partition and which computing node) (3) properties of the component executor threads (thread priority), (4) properties of timers (period and offset), and (5) component interactions and assembly (i.e. the 'wiring').

Figure 2 shows a top-level structure of the CPN-based analysis model. The place *Component Threads* holds a token with a list of all executor threads responsible for component interactions. This list is maintained based on thread priorities on each node so that the highest priority ready thread is always chosen first by the OS scheduler. *Timers* maintains a list of all infrastructural timers in the application. All timer expiries at a specific clock value[1] are handled by the transition *Timer Expiry*. A timer can be used in our component model to trigger the execution of a component operation. DREMS components are dormant by default. Once initialized, a component executor is not eligible to run until there is an operation added to the component message queue. To start a sequence of component interactions, periodic or sporadic timers can be used to trigger an operation of a component.

If a timer triggers a component execution, this component is identified as a candidate for scheduling by *Schedule Thread*. This transition always schedules the highest priority thread that is ready to execute in the active partition on each node. If two threads of equal priority are eligible, the scheduler picks one at random and maintains a round-robin scheduling scheme. If the highest priority thread is not already servicing an operation request, the highest priority operation from its message queue is dequeued and scheduled for execution.

The *Component Message Queues* place is a list that manages the message queues of all components across all nodes. Every time a component thread exe-

[1] The clock values are integers.



(a) Component Assembly (b) Timing Diagram

Fig. 3: Trajectory Planning Application

cutes an operation, the completion of this operation could trigger another component into execution. For instance - the completion of an RMI query on a client component triggers a server-side RMI operation that this server will have to execute. Such interactions are derived from the modeling tools and appropriate tokens are generated in place *Interactions*. When executing component threads, *Execute Thread* checks to see if the execution has any effect on the running thread or on other threads. Therefore, when the client thread completes an RMI query, this thread is moved to *Blocked Threads* and a server RMI operation is placed in *Waiting to Enqueue*. Later, when the server thread is scheduled, the client is unblocked appropriately.

6 State Space Analysis

Given a CPN model (that was generated from a component architecture and deployment model), a state-space of the system can be constructed using the semantics of CPN. This state space is infinite, however, in practice, it is often sufficient to consider some finite subset, starting from a initial state up to a few hyperperiods of the partition scheduler. In order to describe the utility of state space analysis, we consider a simple trajectory planning application (TPA). The component assembly for this application is shown in Figure 3a. A Sensor component periodically publishes on a trigger topic, notifying the Trajectory Planner of the existence of new sensor data. Once the notification is received, the Trajectory Planner makes an RMI call to retrieve the data structure of sensor values, using which the satellite trajectory is updated. The sequence of steps in each of these operations is referred to as the business logic of the operation. This business logic is modeled using a textual language in the modeling tools, in which the designer specifies the macro execution steps in a component operation along with worst-case estimated time taken on each step. Figure 3b shows the expected timing diagram.

The analyzable states of this system are observed in the markings of the various CPN places in the model. Using the built-in state space analysis in *CPN Tools* a bounded state space of the system is generated. Using both standard

and user-defined queries, this state space is searched to check system properties like lack deadline violations and deadlocks, bounds on response times etc.

Deadline Violation Detection: Each time a component operation is scheduled, the clock value of the node is recorded as the "start time" of the operation. If this operation is incomplete when the clock reaches the operation's deadline, a deadline violation is detected. Using the *SearchNodes* function in CPN Tools, the deadline violations on any component operation can be identified by observing all component operations each time the node-specific clock progresses. In Figure 3b, the *DDS_OP* on the Trajectory Planner takes 56 ms to complete, measured from when the operation was enqueued and marked as ready. If the deadline of this operation is set to 50 ms, a state space search would reveal a deadline violation when the clock reaches 51 ms.

Worst-case Trigger-to-Response Time Calculation: For a known trigger operation and desired response operation, the worst-case trigger-to-response time can also be calculated from the generated state space. Using the names of the trigger and response operations, a state space node that presents the earliest completion of the trigger operation and the latest completion of the response operation within the set period is identified. In the Trajectory Planning application, considering the *TIMER_OP* to be the trigger and the trajectory planning *DDS_OP* to be the response, the worst-case response time is found to be 68 ms (Trigger completes earliest at 8 ms and response completes latest at 76 ms).

Partial Thread Execution Order Generation: In development scenarios where an application developer is aware of the operation-specific timing requirements but not thread priorities, the analysis is capable of identifying partial thread execution orders that satisfy the requirements. If all unknown thread priorities are set to a common value, the generated partial state space will then encapsulate the set of non-deterministic thread execution orders that arise from the scheduling. Using timing requirements of the form - *Once Operation A on Component A on Node A completes, Operation B on Component B on Node B must complete within 150 ms*, a state space node satisfying this requirement can be identified by querying the generated state space. A backtrace from this node enables assigning thread priorities to ensure the satisfaction of the timing requirement.

Scalability Testing: The size of the generated state space is dependent on the amount of concurrency in the behavior. If all the executing threads had unique priorities, the thread execution order is a constant as the scheduling is priority-based. However, for large systems with groups of applications and increased concurrency, an equally large state space is required to observe the tree of possible thread executions and operational behaviors. This analysis model has been identified to scale well for medium-sized applications, tested up to 100 components distributed on up to 5 computing nodes. Table 1 summarizes these results.

Table 1: Scalability Results

Scenario	Nodes	Partitions / Node	Threads / Partition	Hyper-periods	State Space	Generation Time
TPA	5	2	1	10	180	0.981s
Sample2	2	5	5	10	124,469	14.1m
Sample3	5	5	4	10	485,552	36.5m

7 Analysis Model Generation

As mentioned in Section 5, the control flow and timing details of component operations are directly integrated into the design-time modeling framework. Using the formal domain-specific model of the system, the configuration of the partition scheduling and component assembly are derived and translated into meaningful CPN tokens. The business logic of each component operation is expressed using a textual language with one attribute per interaction per instance of each component being deployed. Model interpreters parse through this complete design model, instantiating CPN model templates and combining these instances to generate a single integrated .cpn file to analyze the entire system.

8 Future Work

In order to generalize this analysis model and provide flexibility, one possible extension to this approach is to cater to other commonly used scheduling schemes, such as EDF, for component operation scheduling; and novel interaction patterns (e.g. reliable broadcast). Also, the current analysis approach inherits the benefits and the drawbacks of using pessimistic estimates for execution times. Another possible extension to this approach would be to provide a stochastic schedulability analysis allowing for a trade-off between reliability and cost of resources required by the system.

9 Conclusions

Distributed real-time systems operating in dynamic environments, and running mission-critical applications face strict timing requirements to operate safely. This paper presents a Colored Petri Net-based approach to capture the architecture and temporal behavior of such applications for both qualitative and quantitative schedulability analysis. This analysis model includes a compact, scalable representation of high-level design, accounting for the dynamics of real-time thread execution while exploiting knowledge of component execution code. Exhaustive state space search enables verification and validation of useful and necessary system properties, reducing development costs and increasing reliability for such time-critical systems.

Acknowledgments

The DARPA System F6 Program and the National Science Foundation (CNS-1035655) supported this work. Any opinions, findings, and conclusions or rec-

ommendations expressed in this material are those of the authors and do not reflect the views of DARPA or NSF.

References

1. Dubey, A., Emfinger, W., Gokhale, A., Karsai, G., Otte, W., Parsons, J., Szabo, C., Coglio, A., Smith, E., Bose, P.: A Software Platform for Fractionated Spacecraft. In: Proceedings of the IEEE Aerospace Conference, 2012, Big Sky, MT, USA, IEEE (2012) 1–20
2. et al., T.L.: Distributed real-time managed systems: A model-driven distributed secure information architecture platform for managed embedded systems. IEEE Software **31** (2014) 62–69
3. Dubey, A., Gokhale, A., Karsai, G., Otte, W., Willemsen, J.: A Model-Driven Software Component Framework for Fractionated Spacecraft. In: Proceedings of the 5th International Conference on Spacecraft Formation Flying Missions and Technologies (SFFMT), Munich, Germany, IEEE (2013)
4. Otte, W.R., Dubey, A., Pradhan, S., Patil, P., Gokhale, A., Karsai, G., Willemsen, J.: F6COM: A Component Model for Resource-Constrained and Dynamic Space-Based Computing Environment. In: Proceedings of the 16th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC '13), Paderborn, Germany (2013)
5. Jensen, K., Kristensen, L.M.: Coloured Petri Nets - Modelling and Validation of Concurrent Systems. Springer (2009)
6. Renault, X., Kordon, F., Hugues, J.: From aadl architectural models to petri nets: Checking model viability. In: Object/Component/Service-Oriented Real-Time Distributed Computing, 2009. ISORC '09. IEEE International Symposium on. (2009) 313–320
7. Renault, X., Kordon, F., Hugues, J.: Adapting models to model checkers, a case study : Analysing aadl using time or colored petri nets. In: Rapid System Prototyping, 2009. RSP '09. IEEE/IFIP International Symposium on. (2009) 26–33
8. Alpern, B., Schneider, F.B.: Verifying temporal properties without temporal logic. ACM Trans. Program. Lang. Syst. **11** (1989) 147–167
9. Lopez, P., Medina, J., Drake, J.: Real-time modelling of distributed component-based applications. In: Software Engineering and Advanced Applications, 2006. SEAA '06. 32nd EUROMICRO Conference on. (2006) 92–99
10. Harbour, M.G., Garcia, J.J.G., Gutierrez, J.C.P., Moyano, J.M.D.: Mast: Modeling and analysis suite for real time applications. In: In 13th Euromicro Conference on Real-Time Systems. (2001) 125
11. Johnson, K., Calinescu, R., Kikuchi, S.: An incremental verification framework for component-based software systems. In: Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering. CBSE '13, New York, NY, USA, ACM (2013) 33–42
12. ARINC Incorporated Annapolis, Maryland, USA: Document No. 653: Avionics Application Software Standard Inteface (Draft 15). (1997)
13. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77** (1989) 541–580
14. Ratzer, A.V., Wells, L., Lassen, H.M., Laursen, M., Qvortrup, J.F., Stissing, M.S., Westergaard, M., Christensen, S., Jensen, K.: Cpn tools for editing, simulating, and analysing coloured petri nets. In: Proceedings of the 24th International Conference on Applications and Theory of Petri Nets. ICATPN'03, Berlin, Heidelberg, Springer-Verlag (2003) 450–462

Author Index

R

S

V

W