Computers in Industry 151 (2023) 103974

Contents lists available at ScienceDirect

Computers in Industry

journal homepage: www.sciencedirect.com/journal/computers-in-industry

Leveraging the power of formal methods in the realm of enterprise modeling—On the example of extending the (meta) model verification possibilities of ADOxx with Alloy

Sybren de Kinderen [a,*], Qin Ma [b], Monika Kaczmarek-Heß [c]

[a] *Information Systems Group, Eindhoven University Of Technology, The Netherlands*
[b] *Department of Computer Science, University of Luxembourg, Luxembourg*
[c] *Information Systems and Enterprise Modeling Research Group, University of Duisburg-Essen, Germany*

ARTICLE INFO

Keywords:
Enterprise modeling
Enterprise analyses
Meta modeling platforms
ADOxx
Alloy
Verification and consistency checks

ABSTRACT

Verification in the realm of enterprise modeling (EM) ensures both the consistency of EM language specifications (i.e., meta models and additional well-formedness constraints), as well as of enterprise models. The consistency of enterprise models, which integrate different perspectives on an enterprise, ensures that they contain the necessary, in line with domain-specific rules, information for carrying out a variety of model-driven enterprise analyses. Meta modeling platforms are instrumental in carrying out such verification, especially when multiple languages are applied in tandem, as is inherent to enterprise modeling.

This paper reports on our practical experiences of using formal methods for verification in the context of EM. Motivated by the required verification capabilities, we show for one example platform, ADOxx, how it can be chained together with Alloy, an example of lightweight formal method, to capitalize on complementary platform strengths. Namely, ADOxx for language specification and use, and Alloy for verification capabilities. We show the verification, both, on the meta model level, in terms of checking the consistency of language specifications, and on the model level, in terms of checking models against well-formedness constraints. We illustrate the chaining of ADOxx and Alloy on the basis of consistency checks of two languages applied in tandem, namely the value modeling language e3value and the IT infrastructure modeling language, ITML. We also carry out experiments with three further languages to reflect upon the performance of Alloy, and its capability to uncover inconsistencies.

1. Introduction

Enterprise modeling (EM) supports the description, reflection upon, and (re-)design of various aspects of enterprises (e.g., organizational goals, business processes, or IT infrastructure) (Frank, 2014; Sandkuhl et al., 2014). Therefore, EM approaches usually cover multiple perspectives on an organization, modeled using different modeling languages in tandem, and relate these perspectives to each other. A modeling language is typically defined in terms of a meta model and additional constraints (Frank, 2011, p. 3). Being a model of models, this meta model can subsequently be instantiated into models in a dedicated modeling tool.

As one of the main roles of enterprise models is the provision of knowledge on selected aspects within, or related to, an enterprise, they support a variety of analyses (Antunes et al., 2015; Niemann, 2006). Examples include various functional impact-of-change analyses, such as assessing the impact of changing an insurance intermediary on the collaborations it participates (Lankhorst, 2017). Equally, there exist also cost–benefit analysis, compliance analysis, and dependency analysis (Niemann, 2006; Florez et al., 2016). Moreover, due to the multi-perspective nature (Frank, 2014; Sandkuhl et al., 2014), analyses driven by EM require to ensure consistency of different enterprise models, which may be created using different enterprise modeling languages (Jeusfeld, 2016).

A pre-requisite for such model-based analyses is to ensure firstly that the specification of modeling languages (i.e., their meta models and well-formedness constraints) are consistent, and secondly, that enterprise models contain the necessary and correct information, in line with domain-specific rules. This can be partly achieved by verification of a (meta) model, by taking advantage of formal methods (Antunes et al., 2015; Johnson et al., 2007). Formal methods refer to mathematically-based languages, techniques, and tools for specifying and verifying

* Corresponding author.
E-mail address: s.d.kinderen@tue.nl (S. de Kinderen).

https://doi.org/10.1016/j.compind.2023.103974
Received 10 February 2023; Received in revised form 28 April 2023; Accepted 10 June 2023
Available online 18 July 2023

computer systems (Clarke and Wing, 1996). With formal methods, a system is defined in terms of rigorous mathematical entities, and formal analysis techniques can be used to explore the capability of the system and to verify the properties of the system.

Conducting verification of (meta) models is one of the desired functionalities of meta modeling tools, which are used to support the design of (enterprise) modeling methods and the realization of corresponding modeling tools (Ma et al., 2023). Existing studies, e.g., Erdweg et al. (2015), Negm et al. (2019) and Iung et al. (2020), show that meta modeling platforms support the development of software tools, creation of machine-readable models amenable to computer-supported analysis, provide querying capabilities, and more. However, when it comes to support for verification, additional mechanisms are required (Weidmann et al., 2021; Ozkaya and Akdur, 2021; Ma et al., 2023), such as instance generation capabilities for checking the consistency of a language specification.

In response to the above, we show how an example meta modeling platform used often in the EM domain, ADOxx, can be extended with an example lightweight formal method, Alloy, for carrying out verification. This is in line with several existing efforts of using multiple platforms together (Karagiannis and Buchmann, 2018; Jeusfeld, 2016), so as to capitalize on respective platform strengths, as well as with expectations of practitioners (Ozkaya and Akdur, 2021). This paper is a continuation of our earlier work (de Kinderen et al., 2020; Ma et al., 2023) aiming to enhance verification capabilities in the realm of EM. While de Kinderen et al. (2020) focuses on supporting the checking of models against well-formedness constraints only, in this paper, among others: (1) we propose a general solution that can tackle verification of both modeling language specifications (i.e., on the meta model level) and enterprise models (i.e., on the model level); (2) we draw a conceptual mapping between ADOxx and Alloy and define corresponding transformations, and (3) we conduct an experiment with further EM languages, to report on the performance and utility of Alloy.

The paper is structured as follows. Section 2 introduces a motivating scenario, provides a taxonomy of verification mechanisms which meta modeling platforms should provide for EM, and we assess ADOxx, as a representative meta model platform, against this taxonomy to identify its shortcomings. Subsequently, we motivate and demonstrate our ADOxx-Alloy solution in Section 3, and illustrate its feasibility in Section 4. Section 5 evaluates the proposed solution, reports on the performance and utility of the Alloy Analyzer, and discusses related work. Section 6 concludes the paper.

2. Motivating scenario and verification capabilities

As mentioned, one of the aims of EM is to enable enterprise-wide analyses. Such enterprise-wide analyses require the integration of different perspectives, which, in the light of model-driven analyses, requires integration of involved languages.

2.1. Motivating scenario

Across this paper we consider a case, inspired by Gordijn and Akkermans (2005) and further extended based on, e.g., Wang et al. (2019), focusing on interactions between two types of actors in the energy sector: a Distributed Systems Operator (DSO) and consumers equipped with smart meters. The DSO receives high-resolution data from smart meters that provide information on the electricity consumption behavior of the consumers and analyzes them to, among others, prepare personalized load forecasting (Wang et al., 2019). Individual load forecasting constitutes valuable information for customers, as it can be used in their home energy management systems to reduce electricity bills (Keerthisinghe et al., 2018). Therefore, personalized load forecasting is offered by the DSO to customers via an optional payable service termed "metering". To use this service, the customers sign a service contract, and pay a monthly fee. Internally, the DSO realizes the metering service via a metering functionalities software suite running on general-purpose servers.

A model-driven analysis of such a case requires at least the conjoint use of two domain-specific modeling languages (DSMLs), namely a value modeling language (such as e^3*value*, Gordijn and Akkermans, 2001) to capture the value exchanges among involved actors, and an IT infrastructure modeling language (such as the ITML, Frank, 2014) to capture IT infrastructure and associated costs. Integrating e^3*value* and the ITML holds potential for concurrently exploring (1) a value network, (2) the qualities required of an underlying IT infrastructure, and (3) a detailed analysis of cost allocations to IT infrastructure elements. However, modeling language integration is more than simply applying individual languages together. Various relationships, which allow to connect concepts from individual modeling languages and to enable their interaction, should be established during the integration, too. More importantly, consistency of these cross-language relationships should be checked, because relating valid models of individual modeling languages does not necessarily render a valid integrated model (cf. Section 4).

2.2. Verification capabilities required for EM

In addition to checking consistency of cross-language relationships, other verification capabilities are also required for EM. We elaborate on them by constructing a taxonomy of verification capabilities for EM (cf. Fig. 1), following the well-established taxonomy development method of Nickerson et al. (2013).

Following Nickerson et al. (2013), we start by determining the meta characteristic of the taxonomy based on its purpose and expected use. In our case, the taxonomy is mainly to enable discussing (enterprise) meta modeling platforms in terms of their verification capabilities on both the meta model and model levels. Guided by the meta characteristic, taxonomy dimensions are then identified following a conceptual-to-empirical approach. More specifically, we take general (meta) model verification concepts from existing surveys, e.g., Erdweg et al. (2015), Weidmann et al. (2021) and Iung et al. (2020), as the point of departure, and subsequently zoom into those verification concepts relevant to EM, (1) as motivated by our scenario (Section 2.1), and (2) based on known characteristics, such as typical stakeholders involved in EM and the according usability of meta model platforms. Then, in terms of the empirical side, we confront our initial set of dimensions with existing meta modeling platforms and formal methods, and tease out additional required verification capabilities. The resulting taxonomy consists of three main dimensions, explained below.

Intra (meta) model consistency. Within the range of one EM language, this dimension concerns the consistency of both the language specification itself (intra meta model consistency sub-dimension), and of its instances (intra model consistency sub-dimension).

To ensure model consistency one needs to ensure that a model is a valid instance of the modeling language (Weidmann et al., 2021, p. 4). Three main types of constraints needs to be checked to this end: typing, multiplicity, and well-formedness (Weidmann et al., 2021). They follow from the role a modeling language plays, e.g., to define the concepts one can instantiate when building a model and the relations one can establish between those instances. For each relation, the language specifies the types of instances it can connect, referred to as typing constraints, and the number of instances it can connect, referred to as multiplicity constraints. In the mentioned e^3*value* language for example, a value exchange is connected to a value port as input (the in-port) and another value port as output (the out-port). Therefore, both the "in-connects" and "out-connects" relations are defined between a value exchange and a value port, being examples of typing constraints. Moreover, for each value port, one and only one value object should be attached to indicate what is being exchanged. "One and only one" in this context refers to a multiplicity constraint. Moreover, a modeling

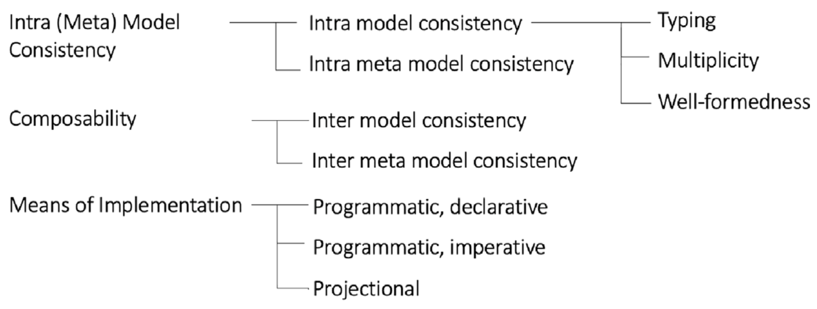

Fig. 1. Taxonomy of verification properties desired from an enterprise modeling perspective.

language can also specify a set of well-formedness constraints (aka. static semantics) to express domain-specific invariants (Strembeck and Zdun, 2009, p. 1261). For example, according to *e*^{3}*value* the value object attached to the in-port of a value exchange must be the same as the value object attached to the out-port of the value exchange.

To ensure modeling language specification consistency, one needs to firstly ensure that the meta model of a modeling language is a consistent instance of its corresponding meta meta model (cf. model consistency above). Secondly, one needs to ensure "no contradiction in its specification, i.e., there is at least one valid instance model" (Semeráth et al., 2017, p. 367). In this regard, instance models one can generate from the language specification provide evidence of its (in)consistency. Thirdly, one needs to also ensure the soundness of the language specification with respect to the domain being captured, namely, that all the instances generated are indeed sensible examples in the domain at hand.

Composability. Due to their multi-perspective nature (Frank, 2014; Sandkuhl et al., 2014) enterprise models usually provide an integrated view covering different diagrams, created using different DSMLs (Frank, 2014). For example, referring to our scenario (Section 2), the integrated consideration of *e*^{3}*value* and ITML enables, for a given value network (modeled in *e*^{3}*value*), exploring detailed cost allocations of the underlying IT infrastructure (modeled in ITML). This dimension concerns the verification of such *composability*, defined as supporting the use of DSMLs together (Erdweg et al., 2012), by systematically adding additional language constructs to the base language, or integrating languages into a single one (Erdweg et al., 2015, p. 8).

Similar to the previous dimension, verification of composability amounts to ensuring consistency of both composed language specifications (the inter meta model consistency sub-dimension) and composed models (the inter model consistency sub-dimension). To ensure inter model consistency, one needs to check if information in the overlapping parts of composed models is conflict free, and that the constraints pertain to their integration are respected across diagrams (Jeusfeld, 2016; Weidmann et al., 2021). To ensure inter meta model consistency, one needs to check if the extra language constructs added to the base language do not contradict the base language, or that overlapping domain knowledge captured in individual DSMLs (that are integrated into a single one) is in accord. For example, following the motivation scenario, an integration between *e*^{3}*value* and ITML will be presented in our case study (Section 4), realized by a bridging relation called "agreed_by". To ensure consistency of the integration, the relation is defined between two specific types, namely the class "Value Object" from *e*^{3}*value* and the class "Service Contract" from ITML. In addition to this typing constraint, multiplicity constraints are also specified for this relation, namely one value object can be agreed by at most one service contract, and vice versa. Moreover, a well-formedness constraint ensuring the agreement of domain knowledge captured in both *e*^{3}*value* and ITML, which is initially overlooked, is discovered by checking the inter meta model consistency. All these three types of constraints should be checked to ensure consistency of composed *e*^{3}*value*–ITML models (i.e., inter model consistency).

In practice, integrated (meta) models are (meta) models themselves, hence inter (meta) model consistency can be checked by the same techniques applied for intra (meta) model consistency.

Means of implementation. This dimension refers to the way in which users of a meta modeling platform can implement the required verification capabilities, which can be programmatic or projectional in nature (Erdweg et al., 2015; Iung et al., 2020).

Implementation of a verification capability is programmatic if it employs a language (Erdweg et al., 2015; Iung et al., 2020), which can be declarative or imperative. Employing a declarative language, one focuses on specifying the verification result to achieve. Examples of declarative languages for this purpose include constraint languages like OCL (OMG, 2014; Gogolla et al., 2007), or logic formalisms. In contrast, employing an imperative language, one focuses on expressing how the desired verification result should be achieved. Here, typically general-purpose programming languages like Java are used (Erdweg et al., 2015; Ma et al., 2015). In contrast, if implementation of a verification capability employs fixed predefined layouts, e.g., in terms of pre-defined forms (Tolvanen and Kelly, 2009), it is projectional (Iung et al., 2020).

On the one hand, usability is an important criterion to compare different means of implementation, whereby the profile of potential users (typical EM community members in our case) should be taken into consideration. Usability concerns both the easiness of specifying and checking of a constraint. On the other hand, the means of implementation (and its underlying formalism) should also be expressive enough to cater for all constraints of interest. According to Levesque (1986) and Levesque and Brachman (1987), there is a trade-off between usability and expressiveness, in the sense that the difficulty of using a language increases dramatically as the expressive power of the language increases.

Generally speaking, projectional mechanisms, like templates or forms, are less powerful compared to programmatic solutions, because the set of possible constraints for the former is pre-defined, hence limited. But, specification of constraints with projectional mechanisms is also in general simpler than programmatic solutions, which either require technical expertise (when it comes to programming language based solutions), or mathematical expertise (when it comes to logic or constraint language based solutions). Note that between programmatic solutions, declarative mechanisms, working at a high level of abstraction, are relatively easier to use than imperative ones.

As it is difficult to decide exactly the level of expressiveness desired in general (because different domains may have different needs of varying complexity), and as profiles and preferences of potential users vary, one should strike a careful balance between aforementioned usability and expressiveness when selecting an appropriate mechanism.

2.3. Verification capabilities of ADOxx

As already mentioned in the introduction, existing studies (Erdweg et al., 2015; Negm et al., 2019; Iung et al., 2020) show that whereas meta modeling platforms offer a plethora of various functionalities relevant to the EM field, they fall short when it comes to verification support (Jeusfeld, 2016; Ozkaya and Akdur, 2021).

To illustrate common shortcomings, we assess the verification capabilities of one representative platform, namely ADOxx, against the dimensions of the taxonomy introduced previously. ADOxx is openly available, and a popular meta modeling platform for developing DSMLs generally (Iung et al., 2020), and for developing DSMLs for the EM field specifically (Ma et al., 2023).

For intra model consistency, ADOxx provides native support to check typing and multiplicity constraints. However, when it comes to checking well-formedness constraints, it falls short. Briefly, ADOxx makes use of AQL queries for this purpose. One needs to execute appropriate queries on a model and interpret query results to decide whether the model is well-formed, both manually. Alternatively, one can also automate the checking by imperatively implementing the checking and interpreting logic in ADOScript. The ADOScript code involves additional complexity, and consequently its use for checking well-formedness can be time consuming, and enhances the risk of introducing errors (de Kinderen et al., 2020).

For intra meta model consistency, according to the taxonomy, a meta model should foremost be a consistent model (instance) of its meta meta model. Meta models are created in ADOxx by instantiating the ADOxx meta meta model. Therefore, basic typing and multiplicity constraints imposed in the meta meta model are respected. However, ADOxx lacks means to detect conflicts in language specifications, i.e., if instances can indeed be created from a specification, and to check the domain soundness thereof, i.e., if created instances are all sensible examples of the domain. Moreover, it even falls short in carrying out a basic sanity check of language specifications as reported in Section 5.3. For composability, although ADOxx explicitly supports cross-diagram modeling, inter (meta) model consistency checking suffers, as a consequence of its deficiency in (1) checking intra model well-formedness, (2) detecting intra meta model conflicts, and (3) assuring intra meta model domain soundness.

Finally, when it comes to means of implementation, ADOxx supports both projectional editing through predefined forms, and imperative programming through ADOScript.

To fill the gaps manifested above, as also remarked by Jeusfeld (2016) and Ozkaya and Akdur (2021), it would be sensible to complement meta modeling platforms with an instrument dedicated to verification such as formal methods, so as to capitalize on their complementary verification strengths. In the following we show how ADOxx, the example meta modeling platform, can be used together with an example light-weight formal method, Alloy, to support verification as desired in the light of the taxonomy.

3. Extending ADOxx with Alloy

In the following, we show how ADOxx can be used in tandem with Alloy, so that the extensive analysis capabilities natively provided by the latter can be exploited. To prepare the discussion, we first summarize the features of the meta modeling platform ADOxx (Section 3.1), and provide a short introduction to Alloy (Section 3.2). Next, we argue for an integration of Alloy in ADOxx in order to leverage Alloy's checking capabilities (Section 3.3). We demonstrate the integration with two scenarios: (1) checking the consistency of, and refining, an EM language (i.e., its meta model and additional well-formedness constraints) developed in ADOxx (Section 3.4); and (2) checking the consistency of enterprise models created in ADOxx against well-formedness constraints (Section 3.5).

3.1. Enterprise modeling language design and usage with ADOxx

Language specifications are implemented with the ADOxx Development Toolkit. For a given language, one defines the abstract syntax (meta model) using the ADOxx Library Language (ALL). In addition by using ADOScript, ADOxx's scripting language, one can define a concrete syntax. Subsequently, the ADOxx Modeling Toolkit is dedicated to language use. It allows for creating models of defined language specifications using the ADOxx Development Language (ADL). Additionally, one can analyze created models by querying them using the ADOxx Query Language AQL.

Fig. 2 summarizes the languages used by the ADOxx approach for creating different components of an enterprise modeling language. The highlighted part is relevant for this work, whereby meta models and models created in ADOxx, using ALL and ADL respectively, are exported in the XML format for interoperation with Alloy to leverage its verification capabilities.

3.2. (Meta) Model checking with Alloy

Alloy (Jackson, 2012) is both a formal language for specifying complex structures, constraints, and behaviors of systems (in terms of Alloy models), and an analyzer for automatically checking properties or simulating the execution of such models. The Alloy specification language is based on first-order relational logic, and can be used for both system specification and constraints definition.

An Alloy model mainly consists of a set of *signatures*, declared with the keyword **sig**. Signatures in Alloy are similar to ADOxx or UML classes. A signature may have zero or many *fields*. Each field defines a relation between the instances of the containing signature and the instances of another signature.[1] Possible instances of an Alloy model, i.e., by populating the signatures and relations, can be controlled by expressing constraints, in two ways: (1) through *multiplicity* constraints given in signature definitions, for which four multiplicities are predefined in Alloy: `lone` for $0..1$, `one` for $1..1$, `set` for $0..*$, and `some` for $1..*$; (2) by defining *facts* that express constraints in terms of logical formulae. If a fact only concerns a single signature, it can be declared right after the signature in the form of a block of logic formulae. This is called a *signature fact*. A signature fact is implicitly universally quantified over the set of instances of the signature, and the logic formulae in the block are joined with logical conjunction.

The Alloy Analyzer generates instances of an Alloy model using a Boolean SAT solver (Torlak and Jackson, 2007). To cope with both finite and infinite Alloy models, the Alloy Analyzer works only within limited scopes relying on the *small scope hypothesis*, namely "a high proportion of bugs can be found by testing a program for all test inputs within some small scope" (Andoni et al., 2003). In other words, analyzing Alloy models within small scopes suffices to unveil most of the bugs in practice. In addition, generated instances of an Alloy model can be displayed in a domain specific visualization, enabling domain experts to review, to confirm, or to point out violation of domain semantics in the instances and subsequently inconsistency in the specified Alloy model (Gammaitoni et al., 2015, 2018; Razo-Zapata et al., 2018).

3.3. Integrating ADOxx with Alloy

To leverage the power of Alloy, an integration of Alloy into both the design and modeling environments of meta modeling platforms would be desired. Indeed, on the one hand, integrating Alloy in the ADOxx Development Toolkit would allow (1) defining well-formedness

[1] Fields in Alloy can define relations of any arity, not only just binary ones as in the case of ADOxx. In this paper, we introduce only the part of Alloy that is relevant for this work.

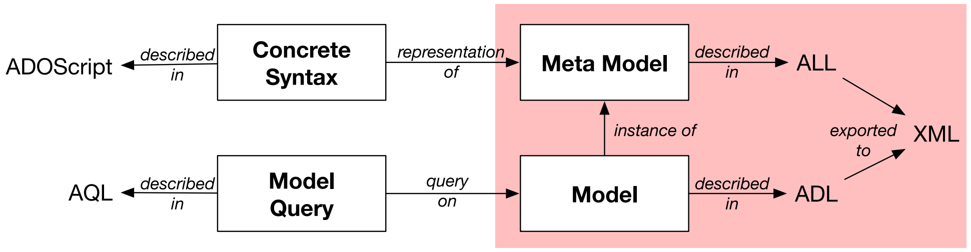

Fig. 2. ADOxx languages.
Source: Adapted from Fill and Karagiannis (2013).

constraints in terms of facts directly in Alloy models corresponding to language specifications, (2) checking consistency of modeling language specifications by analyzing the corresponding Alloy models, and (3) refining modeling language specifications to correct conflicting constraints and/or add missing ones. On the other hand, integrating Alloy in the ADOxx Modeling Toolkit would allow checking and providing informative feedback on the consistency of models.

This paper adopts a loosely coupled integration of ADOxx and Alloy, and chains those platforms together via two model transformations: (1) ADOxxMM2-Alloy to automatically translate meta models developed in ADOxx into Alloy, and (2) ADOxxMdl2Alloy to automatically translate models created in ADOxx into Alloy.

3.3.1. ADOxxMM2Alloy: transforming ADOxx meta models to Alloy models

Meta models are implemented with the ADOxx Development Toolkit using the ADOxx Library Language (ALL). The transformation takes a meta model specification in ALL as input and produces an Alloy model as output. For the formal underpinnings of ALL, especially its formal relation to the Formalism for Describing ADOxx Meta Models and Models (FDMM), we refer to Fill et al. (2013).

We elaborate in the following how different constructs in a meta model specification are transformed.

Classes. ADOxx classes are translated into Alloy signatures. If a class has a superclass specified, it is translated into a subsignature extending the signature corresponding to the superclass.

ALL	`CLASS` ⟨C_1⟩ [: ⟨C_2⟩] `CLASSATTRIBUTE` ⟨ClassAbstract⟩ `VALUE` 0
Alloy	`sig` C_1 [`extends` C_2] {…}

Moreover, an abstract class is translated into an abstract signature.

ALL	`CLASS` ⟨C_1⟩ [: ⟨C_2⟩] `CLASSATTRIBUTE` ⟨ClassAbstract⟩ `VALUE` 1
Alloy	`abstract sig` C_1 [`extends` C_2] {…}

Attributes. An `INTEGER` or `DOUBLE` typed attribute of a class is translated into a field of the signature corresponding to the class. The field has multiplicity `one` and refers to the special predefined signature `Int`, which represents integers.

ALL	`CLASS` ⟨C⟩ `ATTRIBUTE` ⟨att⟩ `TYPE INTEGER` | `DOUBLE`
Alloy	`sig` C {att: `one Int`}

Similarly, an `STRING` or `LONGSTRING` typed attribute in ADOxx is translated into a field with multiplicity `one` and refers to the predefined signature `String`.

ALL	`CLASS` ⟨C⟩ `ATTRIBUTE` ⟨att⟩ `TYPE STRING` | `LONGSTRING`
Alloy	`sig` C {att: `one String`}

An `ENUMERATION` typed attribute in ADOxx is translated into both an Alloy `enum` signature and a field referring to the `enum` signature.

ALL	`CLASS` ⟨C⟩ `ATTRIBUTE` ⟨att⟩ `TYPE ENUMERATION` `FACET` ⟨EnumerationDomain⟩ `VALUE` "e_1@e_2…@e_k"
Alloy	`sig` C {att: `one` att_enum} `enum` att_enum {e_1, e_2,…, e_k}

Finally, an `ENUMERATIONLIST` typed attribute is translated similarly, but with a different multiplicity (assumed by ADOxx), namely `one` for `ENUMERATION` and `some` for `ENUMERATIONLIST`.

ALL	`CLASS` ⟨C⟩ `ATTRIBUTE` ⟨att⟩ `TYPE ENUMERATIONLIST` `FACET` ⟨EnumerationDomain⟩ `VALUE` "e_1@e_2 …@e_k"
Alloy	`sig` C {att: `some` att_enum} `enum` att_enum {e_1, e_2,…, e_k}

Relations. In ADOxx, associations between objects are expressed either as a relationclass or as a special attribute of type `INTERREF`.

An `INTERREF` typed attribute is used to relate an object of the class owning the attribute in one model to an object in another model. As given in Box I, `INTERREF` typed attributes are translated into fields. Note that our transformations ignore the boundary of (meta) models, namely elements of different (meta) models are translated to elements in one "global" Alloy model, because it is orthogonal to the properties we want to check at both the meta model and model levels (cf. Sections 3.4 and 3.5). The lower bound of the multiplicity of an `INTERREF` typed attribute is always 0, while the upper bound is specified by the intValue associated to `max`. If the upper bound is 1, the transformation makes use of the Alloy multiplicity `lone` directly for the field. Otherwise, the Alloy multiplicity `set` is used plus a signature fact specifying the exact bounds.
A relationclass relates objects of a source class to objects of a target class. It is translated into a field of the signature corresponding to the source class.

ALL	`RELATIONCLASS` ⟨R⟩ `FROM` ⟨C_1⟩ `TO` ⟨C_2⟩
Alloy	`sig` C_1{R: `set` C_2}

In contrast to `INTERREF`, one can specify additionally source and target multiplicities of a relationclass R separately from the relationclass definition. As given in Box II, such a multiplicity is defined in the context of a class C using the special class attribute called Class cardinality. One can use `TO-CLASS` to specify the target multiplicity of relation R from class C to class ToC with the lower bound being intValue1 and upper bound intValue2. During the transformation, if intValue1 $\neq 0$ or intValue2 $\neq *$, signature facts are specified for the context class C to explicitly constrain so. Source multiplicity of R from class FromC to class C specified using `FROM-CLASS` is transformed in a similar manner (cf. Box III).
Note that in Alloy, a field f is automatically expanded to `this`.f in signature facts. To access the relation defined by f, one can use the `@` operator. Therefore, #(R & ToC) refers to the number of ToC instances pointed to by a C instance via relation R, and #(`this`.~@R & FromC) refers to the number of FromC instances pointing to a C instance via relation R, whereby & is the set intersection operator and ~f is the reverse of f.

ALL	`CLASS` ⟨C⟩ `ATTRIBUTE` ⟨att⟩ `TYPE INTERREF` `FACET` ⟨AttributeInterRefDomain⟩ `VALUE OBJREF` `mt:`⟨MT⟩ `c:`⟨C⟩′ `max:`⟨intValue⟩
Alloy	`sig` C {att: `lone` C′} — *if* intValue = *1*
	`sig` C {att: `set` C′}{#att >= 0 `and` #att <= intValue} — *otherwise*

Box I.

ALL	`CLASS` ⟨C⟩ `CLASSATTRIBUTE` ⟨Class cardinality⟩ `VALUE RELATION:` ⟨R⟩ `TO-CLASS:` ⟨ToC⟩ `val-min-outgoing:` ⟨intValue1⟩ `val-max-outgoing:` ⟨intValue2⟩
Alloy	`sig` C{. . . }{ #(R & ToC) >= intValue1 — *if* intValue1 $\neq$ *0*; #(R & ToC) <= intValue2} — *if* intValue2 $\neq$ *

Box II.

ALL	`CLASS` ⟨C⟩ `CLASSATTRIBUTE` ⟨Class cardinality⟩ `VALUE RELATION:` ⟨R⟩ `FROM-CLASS:` ⟨FromC⟩ `val-min-incoming:` ⟨intValue1⟩ `val-max-incoming:` ⟨intValue2⟩
Alloy	`sig` C{. . . }{ #(`this`.~@R & FromC) >= intValue1 — *if* intValue1 $\neq$ *0*; #(`this`.~@R & FromC) <= intValue2} — *if* intValue2 $\neq$ *

Box III.

3.3.2. ADOxxMdl2Alloy: transforming ADOxx models to Alloy models

The ADOxx Modeling Toolkit allows creating models (instances) of a defined meta model using the ADOxx Development Language (ADL). This transformation takes a model described in ADL as input and produces also an Alloy model as output. Note that instances of Alloy models cannot be created directly, but are only "generated" via the Alloy Analyzer, because Alloy is not a modeling tool but rather aim to analyze (meta) models. Lacking direct access to instances in Alloy, this transformation capitalizes on the "instance promotion" strategy (Gammaitoni and Kelsen, 2014) to encode a particular instance of an Alloy model in terms of an Alloy model. The result Alloy model, when analyzed by the Alloy Analyzer, has one and only one instance, which corresponds exactly to the original model created in ADOxx. We elaborate in the following how different constructs of a model are transformed.

Objects. An object in an ADOxx model is translated into a singleton Alloy subsignature (i.e., the multiplicity of the signature is **one** hence, can have exactly one atom), extending the signature corresponding to the class of the object (cf. Section 3.3.1).

ADL	`INSTANCE` ⟨o⟩: ⟨C⟩
Alloy	`one sig` o `extends` C{}

Attribute assignments. An attribute assignment of an object is translated into a signature fact associated to the signature corresponding to the object.

ADL	`INSTANCE` ⟨o⟩: ⟨C⟩ `ATTRIBUTE` ⟨att⟩ `VALUE` ⟨value⟩
Alloy	`one sig` o `extends` C{}{att = value}

Links. A link, namely an instantiation of a relationclass, is transformed into a fact associated to the signature corresponding to the source object of the link.

ADL	`INSTANCE` ⟨o_1⟩: ⟨C_1⟩ `INSTANCE` ⟨o_2⟩: ⟨C_2⟩ `CONNECTOR` ⟨r⟩: ⟨R⟩ `FROM` ⟨o_1⟩: ⟨C_1⟩ `TO` ⟨o_2⟩: ⟨C_2⟩
Alloy	`one sig` o_1 `extends` C_1{}{R = o_2} `one sig` o_2 `extends` C_2{}

3.3.3. Prototyping of two transformations

As a proof of concept, we implemented prototypes of these transformations in Python, which take the XML export of meta models defined in ALL (respectively models defined in ADL), and generate their counterparts in Alloy by following the transformation rules given above.

3.4. Leveraging Alloy for language consistency checking

Fig. 3 describes a process to walk through the ADOxx-Alloy integration for meta model consistency checking.

In step 1, a language engineer manually designs a modeling language in ADOxx, exports it into the XML format (`metamodel.xml`), then calls the prototype of ADOxxMM2Alloy transformation (step 2) to produce the corresponding language specification in terms of an Alloy model (`metamodel.als`).

The language engineer continues by manually expressing the set of known domain constraints in terms of Alloy logical formulae in step 3, and launches the Alloy Analyzer to automatically generate all the possible instances of the language specification within a given scope in step 4. If no instance can be generated, this indicates inconsistency in the language specification. A typical example of an inconsistency is that two constraints are conflicting with each other hence, they cannot

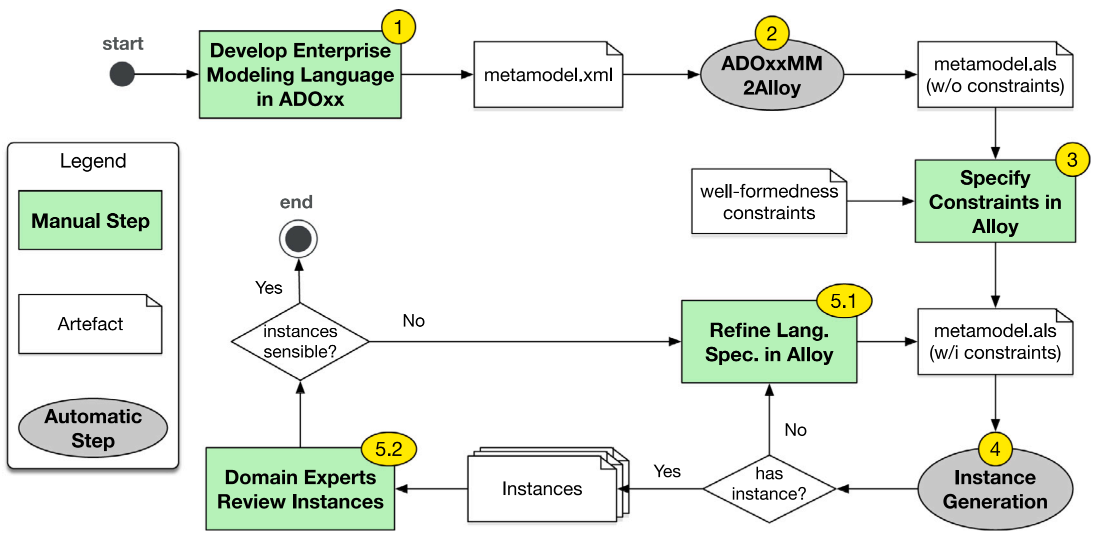
start
Develop Enterprise Modeling Language in ADOxx
1
metamodel.xml
ADOxxMM 2Alloy
2
metamodel.als (w/o constraints)
Legend
Manual Step
Artefact
Automatic Step
well-formedness constraints
Specify Constraints in Alloy
3
end
Yes
instances sensible?
No
Refine Lang. Spec. in Alloy
5.1
metamodel.als (w/i constraints)
No
Domain Experts Review Instances
5.2
Instances
Yes
has instance?
Instance Generation
4

Fig. 3. Chaining ADOxx with Alloy for EM language consistency checking.

be satisfied at the same time. In this case, the language engineer goes to step 5.1 to review, debug, and eventually refine the specification.

Otherwise, if there are indeed instances generated, the current language specification is conflict-free. The language engineer presents the generated instances to a domain expert in step 5.2, and seeks insight in or confirmation of the consistency of the instances with respect to the domain semantics, namely if the generated instances faithfully reflect the domain being captured, e.g., by respecting all the domain rules. The review of generated instances takes the form of a visualization suiting the domain expert. Note that Alloy offers various ways to visualize the generated instances, including a tree view, a table view, and a graphical view. In addition to these, one can also exploit the Alloy-based language workbench Lightning (Gammaitoni et al., 2015) to display generated instances in domain-specific visualizations. As soon as the domain expert detects an anomaly, s/he communicates it to the language engineer. In turn, the language engineer will go to step 5.1 to refine the language specification by adjusting existing and/or adding missing domain constraints. Thereafter, instances will be generated again from the updated language specification and will be reviewed by the domain expert. This continues until all instances are approved by the domain expert.

The refined and approved language specification is used in the next section for checking the consistency of enterprise models. It is also to be reflected in the ADOxx implementation.

3.5. Leveraging Alloy for model consistency checking

This subsection concerns checking the consistency of enterprise models, created in ADOxx, against the modeling language refined and approved in the previous section, using Alloy. In particular, we check if an enterprise model satisfies all the well-formedness constraints. In case elements of an enterprise model violate a well-formedness constraint, we indicate them. Fig. 4 displays a process for using the ADOxx-Alloy integration for model consistency checking.

In step 1, a modeler manually creates, using the ADOxx Modeling Toolkit and modeling languages implemented there, an enterprise model. This model is then exported into XML format (`model.xml`). In step 2, the ADOxxMdl2Alloy transformation takes this XML file, translates it into an Alloy model (`model.als`), and combines it with the Alloy model corresponding to the refined and approved language specification from the previous section (i.e., `metamodel.als` within constraints) with all signatures in `metamodel.als` becoming abstract, and all well-formedness constraints in `metamodel.als` becoming assertions. Recall that the transformation capitalizes on the “instance promotion” strategy (Gammaitoni and Kelsen, 2014) to encode a particular instance of an Alloy model in terms of an Alloy model, because Alloy does not make instances of an Alloy model directly accessible, which are only “generated” via the Alloy Analyzer.

Assertions representing well-formedness constraints are checked in step 3. Checking an assertion in an Alloy model amounts to searching in all possible instances of the Alloy model for an instance that violates the assertion, i.e., a counterexample of the assertion. Recall that the Alloy model (`model.als`) has one and only one instance, which is exactly the enterprise model created in ADOxx whose consistency we want to check. Thus, when checking the well-formedness assertions on the Alloy model, Alloy basically checks if the enterprise model constitutes a counterexample to violate an assertion.

If no assertion is violated, the original enterprise model satisfies all the well-formedness constraints, hence, the model is consistent. Otherwise, if at least one assertion is violated, the model is inconsistent. Moreover, Alloy also binds quantified variables of the assertion to elements of the model, which are responsible for the violation. The modeler exploits this information, debugs the model in ADOxx and when appropriate, s/he repeats the consistency checking process.

4. Illustration: Well-formed and integrated enterprise models

We illustrate the two scenarios using the case introduced in Section 2.

4.1. Consistent integration of e^3value and the ITML

We demonstrate how to shape a consistent integration of *e^3value* and ITML following the chaining of ADOxx and Alloy (Section 3.4). As the purpose is to illustrate how chaining ADOxx with Alloy can support consistency check of (integrated) modeling languages, we focus on a segment of *e^3value* and a segment of ITML only.

4.1.1. Segment of e^3value for value modeling

Fig. 5 shows a simplified value model capturing the value exchanges between the DSO and a consumer. The consumer uses the `Metering` service offered by the DSO as a value object, and pays in return the value object `Metering Fee`. Further, the value exchange is triggered by a `Need for Metering` on the side of the consumer, and the DSO executes the value activity `Provide Metering` to provide the value object `Metering`.

Fig. 6, in the top right corner, presents the meta model of the selected *e^3value* segment. We focus on four concepts `Value Exchange`, `Value Port`, `Value Object`, and `Expense`, and the

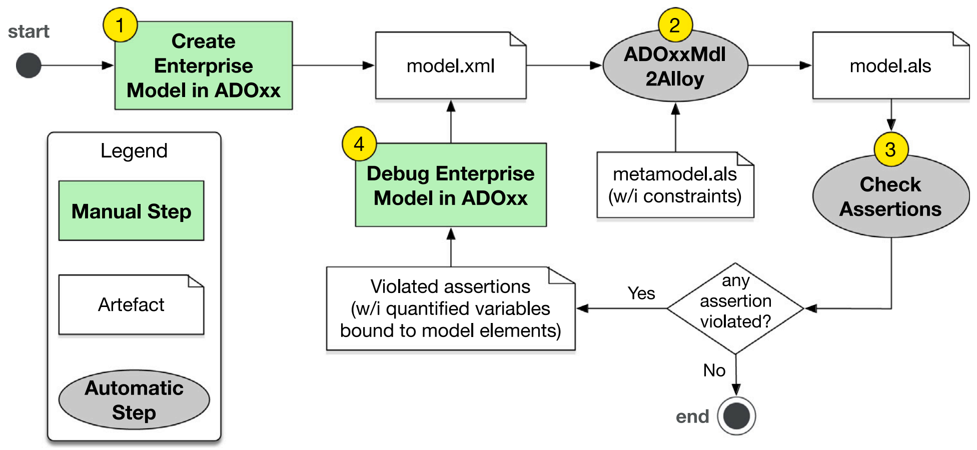
start
1
Create Enterprise Model in ADOxx
model.xml
2
ADOxxMdl 2Alloy
model.als
Legend
Manual Step
Artefact
Automatic Step
4
Debug Enterprise Model in ADOxx
metamodel.als (w/i constraints)
3
Check Assertions
Violated assertions (w/i quantified variables bound to model elements)
Yes
any assertion violated?
No
end

Fig. 4. Chaining ADOxx with Alloy for enterprise models consistency checking.

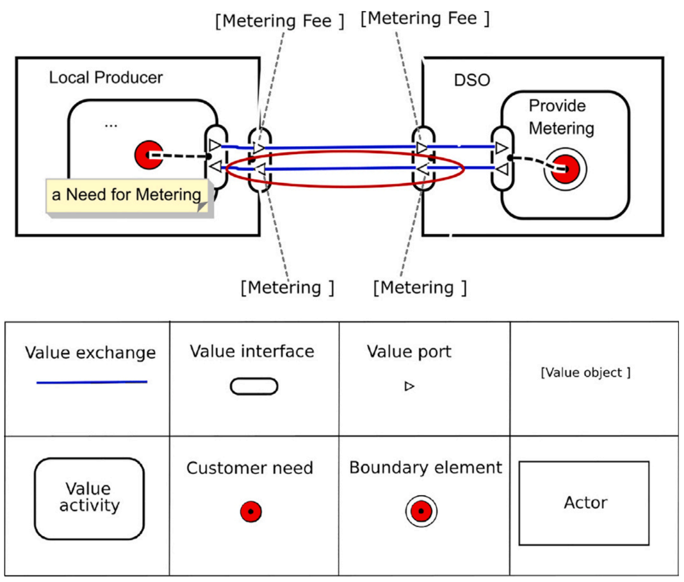

Fig. 5. An example e^3*value* model.

relations among them. A value exchange connects two value ports: one out and one in, each of which has a value object attached. In addition, an expense can be assigned to a value object to indicate its economical value. Expenses form input for e^3*value* profitability calculations.

Moreover, we include one well-formedness constraint (C1) relevant to the selected segment: for a given value exchange, the value object attached to the in-port needs to be the same as the value object of the out-port. Returning to our e^3*value* model (Fig. 5), this constraint requires that the same value object `Metering` is attached to both the in-port and the out-port of the value exchange from the DSO to the consumer.

4.1.2. Segment of ITML for IT infrastructure modeling

Fig. 6, on the left and at the bottom, presents the meta model of the selected ITML segment. A `Service Contract concerns` several `IT Reference Objects`, which is a surrogate for any types of IT infrastructure elements involved in the provision of the service. For the sake of illustration, we include two types of IT elements: `Software` and `Server`. Consider the ITML model in Fig. 7. The `Metering Service Contract`, being a service contract offered by the DSO, involves both a software `Metering Functionalities Software Suite`, and `General Purpose Servers`.

Turning to the cost conception in the ITML, each `IT Reference Object` is associated to a `Cost`, which is subsequently associated to several `Cost Allocations`, to provide a detailed cost breakdown of a service contract. A cost allocation can be subdivided further in different types, of which we focus on two: `Direct Cost Allocation` and `Proportional Cost Allocation`. In our example in Fig. 7, the provision of metering service amounts to (1) purchasing the software `Metering Functionalities Software Suite`, hence incurs a `Direct Cost Allocation` corresponding to the purchasing cost, and (2) the usage of a quota of a server to run the software, hence incurs a `Proportional Cost Allocation` of the total cost of the `General Purpose Servers`. The ratio of the proportional cost allocation, namely 80% in this case, is further justified by an `IT Utilization`, which indicates that the software occupies the server with a very high frequency, hence the (estimated) 80% utilization of the server.

One well-formedness constraint (C2) relevant for the selected ITML segment is included, as defined in Heise (2013). This constraint states that a given cost is either associated to one direct cost allocation, or to several proportional cost allocations, but not a mix of both types of cost allocation.

4.1.3. Consistency checking of an e^3value -ITML integration

To illustrate, we establish in Fig. 6 an example integration of e^3*value* and ITML with a bridging relation between e^3*value* and the ITML called

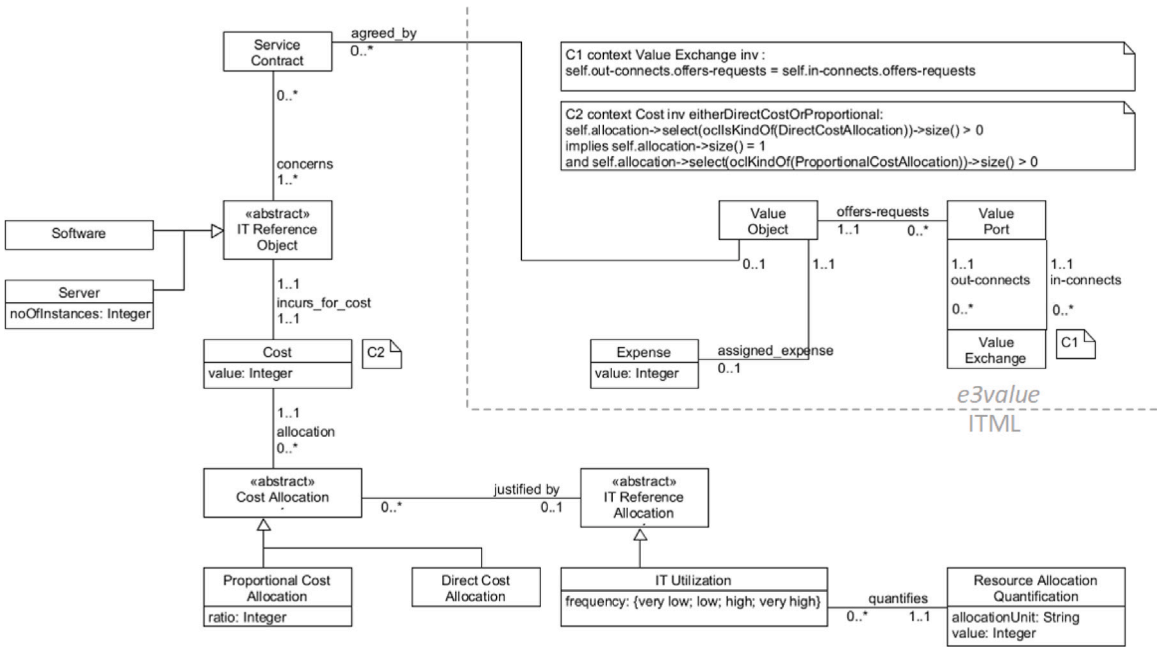
Service Contract
agreed_by
0..*
0..*
concerns
1..*
«abstract» IT Reference Object
Software
Server
noOfInstances: Integer
1..1
incurs_for_cost
1..1
Cost
value: Integer
C2
1..1
allocation
0..*
«abstract» Cost Allocation
justified by
0..*
0..1
«abstract» IT Reference Allocation
Proportional Cost Allocation
ratio: Integer
Direct Cost Allocation
IT Utilization
frequency: {very low; low; high; very high}
quantifies
0..*
1..1
Resource Allocation Quantification
allocationUnit: String
value: Integer
C1 context Value Exchange inv :
self.out-connects.offers-requests = self.in-connects.offers-requests
C2 context Cost inv eitherDirectCostOrProportional:
self.allocation->select(oclIsKindOf(DirectCostAllocation))->size() > 0
implies self.allocation->size() = 1
and self.allocation->select(oclKindOf(ProportionalCostAllocation))->size() > 0
Value Object
offers-requests
1..1
0..*
Value Port
0..1
1..1
1..1
out-connects
0..*
1..1
in-connects
0..*
Value Exchange
C1
Expense
value: Integer
assigned_expense
0..1
e3value
ITML

Fig. 6. A segment of the e^3*value*-ITML integration meta model, implemented in ADOxx, cf. Gordijn (2002, p. 48) and Heise (2013, p. 280).

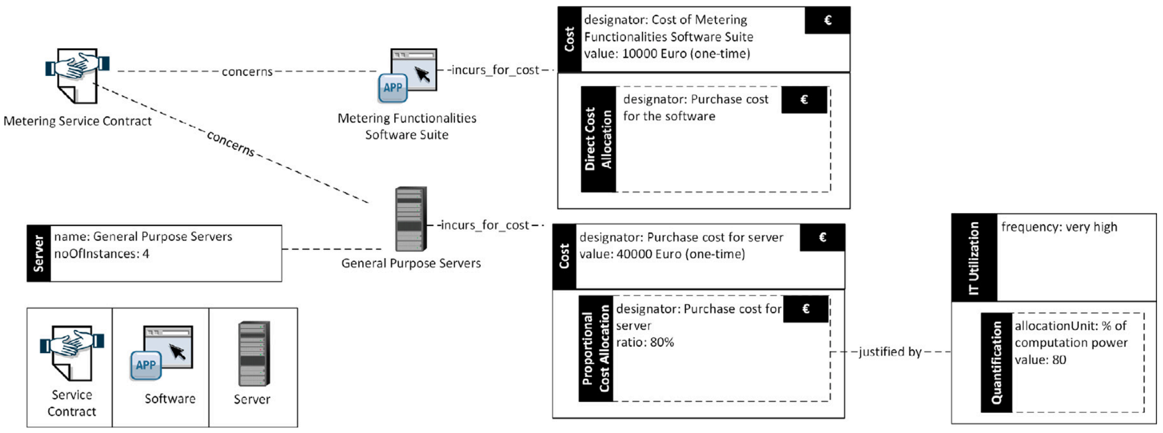
Metering Service Contract
concerns
concerns
APP
Metering Functionalities Software Suite
incurs_for_cost
Cost
designator: Cost of Metering Functionalities Software Suite
value: 10000 Euro (one-time)
€
Direct Cost Allocation
designator: Purchase cost for the software
€
Server
name: General Purpose Servers
noOfInstances: 4
General Purpose Servers
incurs_for_cost
Cost
designator: Purchase cost for server
value: 40000 Euro (one-time)
€
Proportional Cost Allocation
designator: Purchase cost for server
ratio: 80%
€
justified by
IT Utilization
frequency: very high
Quantification
allocationUnit: % of computation power
value: 80
Service Contract
APP
Software
Server

Fig. 7. An example ITML model.

`agreed_by` from `Value Object` to `Service Contract`. Thanks to this relation, a value object offered in an e^3*value* model can be elaborated with further details such as the service contract providing it, the IT infrastructure realizing the service, and the associated cost breakdown thereof, by capitalizing on the modeling capabilities of ITML.

Using the ADOxx Development Toolkit, we implement Fig. 6 in ALL: concepts (shown in Fig. 6 as boxes) are defined as *classes*, relations (shown as lines) as *relationclasses*, and their multiplicities (shown next to the two ends of lines) via the predefined class attribute `Class cardinality`.

In the following, we walk through the tool chain of ADOxx and Alloy as defined in Fig. 3 to illustrate consistency checking of this integration. The ALL definition of the integration is first exported in the XML format (step 1), then transformed into an Alloy model by the ADOxxMM2Alloy transformation defined in Section 3.3.1 (step 2). The resulting Alloy model is given in Appendix A. For example, the ALL definition of the bridging relation `agreed_by` from class `Value Object` in e^3*value* to class `Service Contract` in ITML (given below in the ALL row), is transformed into two signatures (corresponding to the two classes), one field (corresponding to the relationclass), and two signature facts (corresponding to the multiplicities) is given in Box IV.

We then augment the Alloy model with well-formedness constraints originating from e^3*value* and the ITML respectively (step 3). Recall that for the selected segments of e^3*value* and ITML, there are two relevant constraints: C1 from e^3*value* (cf. Section 4.1.1) stating that for a given value exchange, the value object attached to the in-port needs to the same as the value object of the out-port; and C2 from the ITML (cf. Section 4.1.2) stating that a given cost is either associated to one direct cost allocation, or to several proportional cost allocations, but not a mix of both types of cost allocation. We capture C1 by the following fact in Alloy:

```
fact sameValueObjectExchanged{
  all ex: E3_Value_Exchange |
  ex.out_connects.offers_requests =
  ex.in_connects.offers_requests
}
```

and C2 by the following fact:

ALL

```
CLASS ⟨E3_Value_Object⟩
  CLASSATTRIBUTE ⟨Class cardinality⟩ VALUE RELATION: ⟨agreed_by⟩
    TO-CLASS: ⟨ITML_Service_Contract⟩ val-min-outgoing: 0 val-max-outgoing: 1
CLASS ⟨ITML_Service_Contract⟩
  CLASSATTRIBUTE ⟨Class cardinality⟩ VALUE RELATION: ⟨agreed_by⟩
    FROM-CLASS: ⟨E3_Value_Object⟩ val-min-incoming: 0 val-max-incoming: 1
RELATIONCLASS ⟨agreed_by⟩ FROM ⟨E3_Value_Object⟩ TO ⟨ITML_Service_Contract⟩
```

Alloy

```
sig E3_Value_Object{agreed_by: set ITML_Service_Contract}{
  #(agreed_by & ITML_Service_Contract) <= 1}
sig ITML_Service_Contract{}{
  #(this.~@agreed_by & E3_Value_Object) <= 1}
```

Box IV.

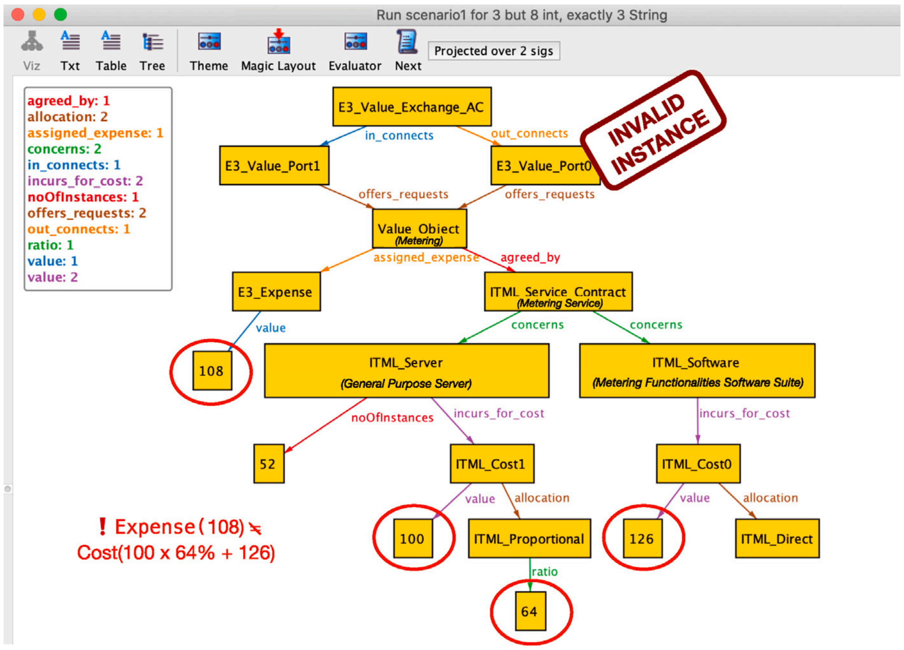

Fig. 8. An instance of the integrated e^3*value*–ITML meta model, generated by Alloy Analyzer.

```
fact eitherDirectCostOrProportional{
  all c: ITML_Cost |
  some c.allocation & ITML_Proportional_Cost_Allocation
  implies
    no c.allocation & ITML_Direct_Cost_Allocation and
  some c.allocation & ITML_Direct_Cost_Allocation implies
    (no c.allocation & ITML_Proportional_Cost_Allocation
    and one c.allocation)
}
```

These two facts restrict the Alloy Analyzer to only search for instances of integrated e^3*value*–ITML meta model, of which both the e^3*value* part and ITML part are valid e^3*value* and ITML models respectively.

As a result, the Alloy Analyzer generates multiple instances (step 4). These instances are reviewed (step 5.2) and one inconsistent instance (as shown in Fig. 8) is noticed. Specifically, in this instance an expense is assigned to the value object. The same value object is also agreed by a service contract providing the value object. Realizing the service concerns the usage of a server and a software, which incurs both the cost of purchasing the software, modeled as a direct cost allocation of value 126, and the cost of using the server, modeled as a proportional cost allocation of the total cost of the server, namely 100, with a ratio of 64%.[2] Therefore, the total cost to offer the value object is $100 \times 64\% + 126 = 190$, i.e., the expense of the value object. However, in the generated instance the expense assigned to the value object is $108 \neq 190$. This anomaly is due to an inconsistency in the e^3*value*–ITML integration. We correct it by adding a new well-formedness constraint C3: the expense assigned to a value object in e^3*value* should be equal to the costs incurred for offering this value object elaborated in the ITML. Note that C3 is only relevant when e^3*value* and ITML are used in tandem. The following Alloy fact captures C3:

```
fact ExpenseEqualsCost{
  all vo: E3_Value_Object |
    some vo.agreed_by and some vo.assigned_expense
    implies vo.assigned_expense.value =
      (sum c: vo.agreed_by.concerns.incurs_for_cost |
      actualCost[c])
}

fun actualCost[c: ITML_Cost]: Int{
  some c.allocation & ITML_Proportional_Cost_Allocation
    implies mul[c.value, c.allocation.ratio] else c.value
}
```

[2] Note that costs generated for the instance are only indicative and do not necessarily correspond to real costs.

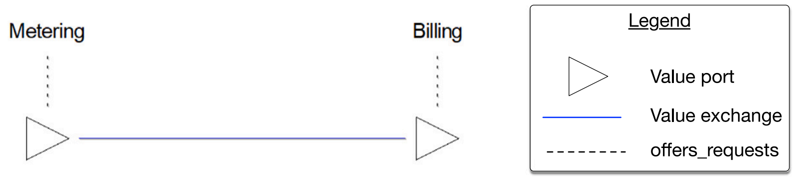

Fig. 9. An example ill-formed e^3*value* model, created with ADOxx.

ADL	`INSTANCE` ⟨E3_Value_Port_105603⟩: ⟨E3_Value_Port⟩ `INSTANCE` ⟨Metering⟩: ⟨E3_Value_Object⟩ `CONNECTOR` ⟨r⟩: ⟨offers_requests⟩ `FROM` ⟨E3_Value_Port_105603⟩: ⟨E3_Value_Port⟩ `TO` ⟨Metering⟩: ⟨E3_Value_Object⟩

Alloy	`one sig` E3_Value_Port_105603 `extends` E3_Value_Port{}{offers_requests = Metering}
	`one sig` Metering `extends` E3_Value_Object{}

Box V.

4.2. *Well-formedness checking of* e^3*value models*

Now we zoom into the e^3*value* part of the case to illustrate consistency checking of models following the chaining of ADOxx and Alloy (Section 3.5). Recall that ADOxx natively checks both (1) typing constraints, e.g., a value port can be related to a value object via the `offers-requests` relation, and (2) multiplicity constraints, e.g., a value port can be related to exactly one value object via the `offers-requests` relation. But it falls short in checking well-formedness constraints.

Fig. 9 shows an example e^3*value* model instantiating the e^3*value* segment in Fig. 6. We implement the model in ADL using ADOxx Modeling Toolkit. In the following, we walk through the tool chain of ADOxx and Alloy presented in Fig. 4 to illustrate well-formedness checking of this example model. The ADL definition of the example e^3*value* model is first exported in the XML format (step 1), then transformed into an Alloy model by the ADOxxMdl2Alloy transformation defined in Section 3.3.2 (step 2). The resulting Alloy model is given in Appendix B. For example, the ADL definition, given below in the ADL row, of the left value port in Fig. 9 and the `Metering` value object related to it via a `offers_requests` link, is transformed into two singleton sub signatures extending the signatures corresponding to their classes respectively, and one fact in the signature corresponding to the value port instance is given in Box V.
We proceed to check if this model is well-formed. To do so, we add a command in the Alloy model to check the assertion representing the well-formedness constraint C1 (step 3).

```
// Constraint C1 from e3value
assert sameValueObjectExchanged{
     all ex: E3_Value_Exchange |
          ex.out_connects.offers_requests =
          ex.in_connects.offers_requests
}

check sameValueObjectExchanged
```

Fig. 10 shows a counterexample found by the Alloy Analyzer corresponding to the e^3*value* model in Fig. 9. The box on top, which represents the value exchange instance, has an additional label (highlighted in red). This label indicates that if one lets the variable `ex: E3_Value_Exchange` defined in the assertion `sameValueObjectExchanged` above be the value exchange object named `E3_Value_Exchange_105621` of the example model, the very assertion is violated, because the in-port and the out-port of this value exchange have two different value objects attached, namely `Metering` and `Billing` respectively. In other words, the e^3*value* model in Fig. 9 is not well-formed: it violates the well-formedness constraint C1 represented by the assertion.

5. Evaluation and discussion

In this section, we discuss the ADOxx-Alloy tool chain's verification capabilities, report on the experience of using the Alloy Analyzer, and consider related approaches.

5.1. *ADOxx-Alloy and its verification capabilities*

A key takeaway is that Alloy's verification capabilities provide a nice complement to ADOxx, on both the meta model level, as well as on the model level. On the meta model level, Alloy's native instance generation facilities allow a domain expert to check instance sensibility. As such, a domain expert can check the sensibility of a language specification by observing, e.g., conflicts or inconsistencies that need to be resolved. On the model level, Alloy offers natively model checking capabilities, allowing for a check of a model against a set of well-formedness constraints. Additionally ADOxx and Alloy both support composability, as also showcased with our illustrative scenario. This composability is a necessary prerequisite for the kind of cross-model and cross-language analysis inherent to the realm of enterprise modeling.

Furthermore, when it comes to means of implementation, compared to other (meta-)model checkers Alloy is attractive as it combines expressiveness with a relative usability. Specifically, in terms of usability Alloy not only provides a substantive feedback to users on how and why (e.g., in terms of what well-formedness constraints were violated), but also offers ease-of-use based on our subjective experience. The ease-of-use of Alloy is echoed by several related works. For one, in relating UML to Alloy (Anastasakis et al., 2010) focus on Alloy in contrast to theorem provers such as KeY (a theorem prover focusing on the formalization of OCL, Ahrendt et al., 2005) which, as declared by Anastasakis et al. (2010), require special expertise. Similarly, in relating UML to Alloy various works e.g., Braga et al. (2010) and Cunha et al. (2015), emphasize Alloy's lightweight nature, with Cunha et al. (2015) particularly comparing favorably its inherently user friendly analysis tool to "heavy-weight" theorem provers requiring a steep learning curve.

Finally, in case users would prefer to formulate constraints in OCL instead, this would also be possible, as approaches exist allowing for (automatic) transformation of OCL statements into Alloy, e.g., Cunha et al. (2015).

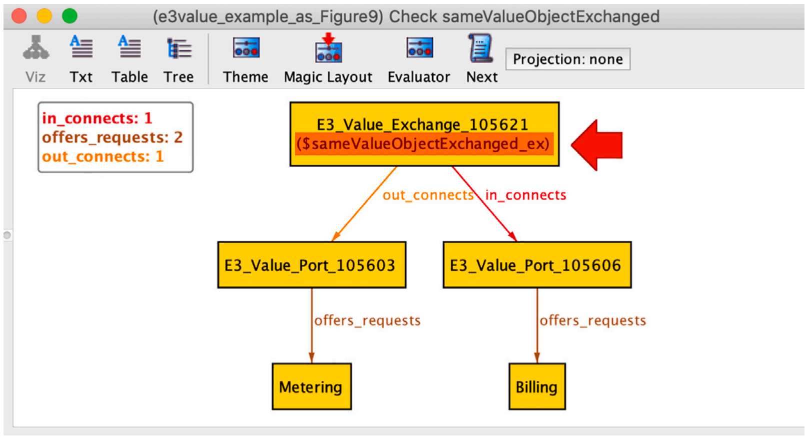

Fig. 10. The value exchange that violates the well-formedness constraint indicated by Alloy Analyzer.

Table 1
Analysis time of Alloy[a] for the case study.

Alloy model	Scope	Time (s)
Listing Appendix A[b]	3 but exactly 3 String, 8 Int	0.196
Listing Appendix B	Default scope[c]	0.002

[a]Alloy Analyzer version 5.1.0, Solver=sat4j, measured on a 2,3 GHz 8-Core Intel Core i9 CPU with 768 MB memory.
[b]Extended with additional predicates to avoid trivial empty instances.
[c]The scope is irrelevant in this case, because of the "single instance" property of ADOxxMdl2Alloy transformation (cf. Section 3.5).

5.2. Performance

We report on the performance of the Alloy Analyzer, when it is applied to the case (cf. Table 1), and when it is applied to analyzing other sample enterprise modeling languages with a different complexity (cf. Table 2). Specifically, the first row of Table 1 expresses that within a small scope (i.e., 3 but exactly 3 String, 8 Int) and within a short period of time (i.e., 0.196 s), the Alloy Analyzer is already able to generate a to-the-point instance of the integration of e^3*value* and ITML (whereby Listing Appendix A is the corresponding Alloy model of the integration). This helps uncover a missing well-formedness constraint that is only relevant when the two languages are integrated. The second row shows the time (i.e., 0.002 s) used to check the well-formnedness of an example e^3*value* model (cf. Fig. 9 & Listing Appendix B).

Bearing in mind that a proposed solution is useful only if it can scale beyond relatively small examples, we experiment further with the ADOxx-Alloy tool chain on other sample modeling languages used in the EM domain. Specifically, we test (1) an extended version of e^3*value* to include also the notions around "market segment", (2) the Extended Entity-Relationship model (EER), and (3) the Business Process Model and Notation (BPMN 2.0). The ADOxx implementations of EER and BPMN 2.0 can be downloaded from the ADOxx website.[3] These further experiments are to allow for (1) testing the general applicability of the ADOxx-Alloy tool chain, (2) testing the performance while scaling up the complexity of languages being checked and the size of the state space being searched, and last but not least (3) drawing lessons learned from the combined use of ADOxx and Alloy.

The complexity of a language specification is at least influenced by the following factors: the number of concepts (signatures in Alloy), the number of relations (fields in Alloy), and the number of constraints (facts in Alloy). Moreover, given a language specification, the Alloy

[3] EER: https://www.adoxx.org/live/er, BPMN 2.0: https://www.adoxx.org/live/bpmn.

Table 2
Analysis time of Alloy[a] for sample EM languages.

EM Lang.	Signature#	Field#	Fact#	Scope	Time[b] (s)
e^3*value*	27	26	12	30	0.239
				40	0.711
				50	0.996
				60	1.358
				70	3.007
				80	2.969
EER[c]	154	80	0	100	0.358
				110	0.408
				120	0.532
				130	0.567
				140	0.630
				150	0.691
BPMN 2.0[c]	443	605	290	20	0.681
				30	1.394
				40	2.917
				50	4.551
				60	7.207
				70	19.593

[a]Alloy Analyzer version 5.1.0, Solver=sat4j, measured on 2,3 GHz 8-Core Intel Core i9 CPU with 3 GB memory.
[b]Average of 3 executions.
[c]Language realization available on the ADOxx website: https://www.adoxx.org/live/implementation-cases.

Analyzer can also search in spaces of different sizes (defined by the scope) for satisfying instances of a predicate, or counterexamples of an assertion. Note that if no satisfying instance of a predicate can be found within a scope, this does not imply the predicate is inconsistent. Also, when no counterexample of an assertion can be found within a scope, this does not imply that the assertion is valid, because satisfying instances or counterexamples may just beyond the given scope. Therefore, in case large complex languages need to be analyzed, one typically starts by analyzing in a small search space and expands it gradually, until either useful evidence are found, or the limit is reached (which is set by memory size, analysis time, and/or the Alloy Analyzer translation capacity). Indeed, model finders like Alloy perform only incomplete analyses, compared with model checkers or theorem provers (Jackson, 2012), and this is where we rely on the "small scope hypothesis".

Table 2 summarizes the results of experimenting on other sample EM languages. For each language, we also take note of the number of signatures, the number of fields, and the number of facts in the corresponding Alloy model of the language (which is produced by the ADOxxMM2Alloy transformation, cf. Section 3.4), as an indicator of the complexity of the language. Moreover, analysis starts from a scope, and scales up from there with a step of 10.

5.3. Reflection on the use formal methods

Our experience shows that Alloy, relative to ADOxx, offers more powerful type checking capabilities on the meta model level. Especially, this expresses itself in the following two cases.

Sanity checking of constraints specified on relations defined within an inheritance hierarchy. During our experiment with the standard BPMN 2.0 implementation of ADOxx, we noticed a general relationclass called "Message Flow" defined between D-Construct and D-Construct, with D-Construct being a pre-defined root class in ADOxx. Then for each subclass of D-Construct for which the relation does not hold, a cardinality constraint of value 0..0 is specified to rule out the instantiation of the relation between instances of the given subclass. Somehow, the ADOxx BPMN 2.0 implementation incorrectly specifies such cardinality constraints for classes that do not inherit from D-Construct, too, and ADOxx fails to detect the mistake. Differently, after applying the ADOxxMM2Alloy transformation to the BPMN 2.0 ADOxx implementation, Alloy issues a warning on the generated BPMN 2.0 Alloy model, complaining that a constraint is specified for a non-existing relation.

Checking of directionality of relations. During our experiment with the e^3*value* implementation of ADOxx, we defined a relationclass from the source class E3_Value_Interface to the target class E3_Actor. We also defined according cardinalities. However, we did so (accidentally) in the wrong direction, i.e., by treating E3_Actor as the source class and E3_Value_Interface as the target class. Remarkably, this was not only allowed by ADOxx, there were even no warnings generated. Differently, again after a transformation to Alloy the inconsistency was directly pointed by the Alloy Analyzer.

5.4. Other approaches to extend ADOxx with verification capabilities

Different examples exist that conjointly use meta modeling platforms and formal method tools, e.g., Anastasakis et al. (2010), Maoz et al. (2011), Semeráth et al. (2017) and Kuhlmann et al. (2011). Specifically, when it comes to *extending ADOxx with verification capabilities*, also additional initiatives exist, e.g., Jeusfeld (2016), Karagiannis et al. (2016) and Karagiannis and Buchmann (2018). Whereas most of those initiatives capitalize on RDF-based representation and propose the use of ADOxx together with GraphDB, e.g., Karagiannis and Buchmann (2018) to benefit from the graph-based inference of the latter, or Karagiannis et al. (2016) who capitalizes on SPARQL queries over RDF graphs to achieve among others inter model consistency; the closest to our effort seems to be the work from the SemCheck project (Jeusfeld, 2016). The proposed integration architectures of our work and SemCheck are similar: both connect ADOxx with an external platform, i.e., Alloy or ConceptBase, via an intermediate, i.e., ADOxx to Alloy transformations or ADOxx/Telos Adapter. This allows for translation of ADOxx (meta) models to the target platform in terms of the Alloy or the ConceptBase syntax.

The difference between Jeusfeld (2016) and our approach exhibits itself in the connected platforms. It is difficult to draw a judgment between the two, as both have their strengths and weaknesses. For example, ConceptBase focuses more on internal model consistency (Jeusfeld, 2016), i.e., to check if a (meta) model conforms to a (meta) meta model. It addresses less so external model verification, i.e., the check of (dynamic) semantics of (meta) models. In contrast, as demonstrated in Section 3.4, a mainstream use case scenario of Alloy is to verify the integrity of meta models by automatically generating instances of partial specifications and validating the semantic consistency of the generated instances. This allows for detecting abnormal instances and subsequently completing/correcting meta model definitions (cf. Section 4.1.3 and Gammaitoni et al., 2015). Moreover, Alloy also supports model execution simulation and checking properties of such execution traces (Kelsen and Ma, 2008).

In addition, different non-functional properties of the connected platforms are a factor to consider, e.g., being intuitive to use, easy to learn, efficient to reason, and popular in education. While users' preference between the syntax used by the two target platforms is more of a subjective matter, one difference is worth noticing. Alloy uses an unified language for both (meta) model and constraints specification, while ConceptBase uses two separate languages: (1) Telos frames to capture (meta) models, and (2) Datalog (which is based on first-order logic) formulae to define queries and deductive rules to check consistency of those models. Moreover, although both ConceptBase and Alloy are based on first-order logic, in contrast to Alloy where transitive closure is provided as the first class citizen, transitivity is not provided natively by ConceptBase. Rather, it requires the user to define it in terms of deductive rules.

6. Conclusions

In this paper, we demonstrate how one selected meta modeling platform, ADOxx, can be meaningfully complemented by an example lightweight formal method, Alloy, for two types of checks: (1) on the meta model level, where we check the consistency of modeling language specifications by capitalizing on the instance generation capabilities of Alloy; and (2) on the model level, where we check models against well-formedness constraints of modeling language specifications by capitalizing on assertions. We illustrate our solution with a case from the energy sector, involving the integrated use of two EM languages, and report on experiments with three further EM languages, to reflect both upon the performance of Alloy in terms of "scaling up", and upon the added value of its verification capabilities as a formal method.

When it comes to limitations, during the meta model consistency check of the case we introduced the following simplifications. Firstly, only indicative values (which are smaller than the real-life values) for the costs of the software and the server are used, to cope with Alloy's instance generation getting increasingly resource intensive when the scope is increased. Secondly, because Alloy natively lacks data types for representing floating point numbers, we have changed all the attributes of the floating point type to attributes of the integer type in the meta model. These simplifications did not affect the meta model consistency check, as the inconsistency is orthogonal to the actual values and types of costs, and Alloy found the major inconsistency to-be-addressed in the e^3*value*-ITML integration. Still for future work we need to keep this limitation in mind.

When it comes to future work, in this paper we illustrate the chaining of ADOxx and Alloy using a loosely coupled strategy via transformations from ADOxx to Alloy. We plan to also investigate deeper integration of the two platforms. On the model level, we consider on-the-fly well-formedness checking and consistency checking. As a first step we could experiment with the Olive microservices framework, developed by OMiLAB to support modular meta modeling platform development and dissemination (Bork et al., 2019, pp. 683–684). On the meta model level, it would be interesting to extend the transformations to become bi-directional, namely not only from ADOxx to Alloy, but also from Alloy back to ADOxx to facilitate the amendment/refinement of modeling language specifications on the side of ADOxx, based on verification results provided by Alloy.

CRediT authorship contribution statement

Sybren de Kinderen: Conceptualization, Methodology, Investigation, Writing – original draft, Writing – review & editing, Software. **Qin Ma:** Conceptualization, Methodology, Investigation, Writing – original draft, Writing – review & editing, Software. **Monika Kaczmarek-Heß:** Conceptualization, Methodology, Investigation, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.compind.2023.103974.

References

Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., et al., 2005. The key tool. Softw. Syst. Model. 4 (1), 32–54.

Anastasakis, K., Bordbar, B., Georg, G., Ray, I., 2010. On challenges of model transformation from UML to Alloy. Softw. Syst. Model. 9 (1), 69–86.

Andoni, A., Daniliuc, D., Khurshid, S., 2003. Evaluating the "Small Scope Hypothesis". Tech. Rep., MIT-LCS-TR-921, MIT CSAIL.

Antunes, G., Barateiro, J., Caetano, A., Borbinha, J., 2015. Analysis of federated enterprise architecture models. In: ECIS 2015 Completed Research Papers. Paper 10.

Bork, D., Buchmann, R., Karagiannis, D., Lee, M., Miron, E.-T., 2019. An open platform for modeling method conceptualization: The OMiLAB digital ecosystem. Commun. Assoc. Inf. Syst. 44, 673–697.

Braga, B.F., Almeida, J.P.A., Guizzardi, G., Benevides, A.B., 2010. Transforming OntoUML into alloy: towards conceptual model validation using a lightweight formal method. Innov. Syst. Softw. Eng. 6 (1), 55–63.

Clarke, E.M., Wing, J.M., 1996. Formal methods: State of the art and future directions. ACM Comput. Surv. 28 (4), 626–643.

Cunha, A., Garis, A.G., Riesco, D., 2015. Translating between alloy specifications and UML class diagrams annotated with OCL. Softw. Syst. Model. 14 (1), 5–25.

Erdweg, S., Giarrusso, P.G., Rendel, T., 2012. Language composition untangled. In: Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications. pp. 1–8.

Erdweg, S., van der Storm, T., Völter, M., Tratt, L., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G., Molina, P.J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V., Visser, E., van der Vlist, K., Wachsmuth, G., van der Woning, J., 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. Comput. Lang. Syst. Struct. 44, 24–47.

Fill, H., Karagiannis, D., 2013. On the conceptualisation of modelling methods using the ADOxx meta modelling platform. EMISA 8 (1), 4–25.

Fill, H.-G., Redmond, T., Karagiannis, D., 2013. Formalizing meta models with FDMM: the ADOxx case. In: Enterprise Information Systems: 14th International Conference, ICEIS 2012, Wroclaw, Poland, June 28-July 1, 2012, Revised Selected Papers 14. Springer, pp. 429–451.

Florez, H., Sánchez, M., Villalobos, J., 2016. A catalog of automated analysis methods for enterprise models. SpringerPlus 5 (1), 406.

Frank, U., 2011. The MEMO Meta modeling Language (MML) and Language Architecture, ICB-Research Report 43, second ed. University of Duisburg-Essen, Essen.

Frank, U., 2014. Multi-perspective enterprise modeling: Foundational concepts, prospects and future research challenges. Softw. Syst. Model. 13 (3), 941–962.

Gammaitoni, L., Kelsen, P., 2014. Domain-specific visualization of Alloy instances. In: Proceedings of the 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ 2014). In: LNCS, vol. 8477, pp. 324–327.

Gammaitoni, L., Kelsen, P., Glodt, C., 2015. Designing languages using lightning. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2015). pp. 77–82.

Gammaitoni, L., Kelsen, P., Ma, Q., 2018. Agile validation of model transformations using compound F-alloy specifications. Sci. Comput. Program. 162, 55–75.

Gogolla, M., Büttner, F., Richters, M., 2007. USE: A UML-based specification environment for validating UML and OCL. Sci. Comput. Program. 69 (1), 27–34.

Gordijn, J., 2002. Value-Based Requirements Engineering: Exploring Innovatie E-Commerce Ideas (Ph.D. thesis). Vrije Universiteit Amsterdam.

Gordijn, J., Akkermans, J., 2001. e3-value: Design and evaluation of e-business models. IEEE Intell. Syst. 11–17.

Gordijn, J., Akkermans, H., 2005. Business models for distributed energy resources in a liberalized market environment. Electr. Power Syst. Res. J. 77 (9), 1178–1188.

Heise, D., 2013. Unternehmensmodell-Basiertes IT-Kostenmanagement Als Bestandteil Eines Integrativen IT-Controllings. Logos, Berlin.

Iung, A., Carbonell, J., Marchezan, L., Rodrigues, E., Bernardino, M., Basso, F.P., Medeiros, B., 2020. Systematic mapping study on domain-specific language development tools. Empir. Softw. Eng. 25 (5), 4205–4249.

Jackson, D., 2012. Software Abstractions: Logic, Language, and Analysis, revised ed. The MIT Press.

Jeusfeld, M.A., 2016. SemCheck: Checking constraints for multi-perspective modeling languages. In: Karagiannis, D., Mayr, H.C., Mylopoulos, J. (Eds.), Domain-Specific Conceptual Modeling: Concepts, Methods and Tools. Springer, pp. 31–53.

Johnson, P., Lagerström, R., Närman, P., Simonsson, M., 2007. Enterprise architecture analysis with extended influence diagrams. Inf. Syst. Front. 9 (2–3), 163–180.

Karagiannis, D., Buchmann, R.A., 2018. A proposal for deploying hybrid knowledge bases: the ADOxx-to-GraphDB interoperability case. In: Proceedings of the 51st HICSS.

Karagiannis, D., Buchmann, R., Bork, D., 2016. Managing consistency in multi-view enterprise models: An approach based on semantic queries. In: 24th European Conference on Information Systems (ECIS 2016).

Keerthisinghe, C., Verbič, G., Chapman, A.C., 2018. A fast technique for smart home management: ADP with temporal difference learning. IEEE Trans. Smart Grid 9 (4), 3291–3303.

Kelsen, P., Ma, Q., 2008. A lightweight approach for defining the formal semantics of a modeling language. In: Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2008). In: LNCS, vol. 5301, Springer, pp. 690–704.

de Kinderen, S., Ma, Q., Kaczmarek-Heß, M., 2020. Towards extending the validation possibilities of ADOxx with Alloy. In: Grabis, J., Bork, D. (Eds.), The Practice of Enterprise Modeling. Springer, pp. 138–152.

Kuhlmann, M., Hamann, L., Gogolla, M., 2011. Extensive validation of OCL models by integrating SAT solving into USE. In: Bishop, J., Vallecillo, A. (Eds.), Objects, Models, Components, Patterns. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 290–306.

Lankhorst, M., 2017. Enterprise Architecture at Work: Modeling, Communication and Analysis, fourth ed. Springer.

Levesque, H.J., 1986. Knowledge representation and reasoning. Annu. Rev. Comput. Sci. 1 (1), 255–287.

Levesque, H.J., Brachman, R.J., 1987. Expressiveness and tractability in knowledge representation and reasoning 1. Comput. Intell. 3 (1), 78–93.

Ma, Q., Kaczmarek-Heß, M., de Kinderen, S., 2023. Validation and verification in domain-specific modeling method engineering: an integrated life-cycle view. Softw. Syst. Model. 22 (2), 647–666.

Ma, Q., Kelsen, P., Glodt, C., 2015. A generic model decomposition technique and its application to the Eclipse modeling framework. Softw. Syst. Model. 14 (2), 921–952.

Maoz, S., Ringert, J.O., Rumpe, B., 2011. CD2alloy: Class diagrams analysis using Alloy revisited. In: Whittle, J., Clark, T., Kühne, T. (Eds.), 14th International MODELS Conference, Wellington, New Zealand, October 16-21, 2011. Proceedings. In: LNCS, vol. 6981, Springer, pp. 592–607.

Negm, E., Makady, S., Salah, A., 2019. Survey on domain specific languages implementation aspects. Int. J. Adv. Comput. Sci. Appl. 10 (11).

Nickerson, R.C., Varshney, U., Muntermann, J., 2013. A method for taxonomy development and its application in information systems. Eur. J. Inf. Syst. 22 (3), 336–359.

Niemann, K.D., 2006. From Enterprise Architecture to IT Governance, Vol. 1. Springer.

OMG, 2014. Object Constraint Language (OCL), Version 2.4. Tech. Rep., URL https://www.omg.org/spec/OCL/2.4/PDF.

Ozkaya, M., Akdur, D., 2021. What do practitioners expect from the meta-modeling tools? A survey. J. Comput. Lang. 63, 101030.

Razo-Zapata, I.S., Chew, E., Ma, Q., Gammaitoni, L., Proper, H.A., 2018. Enabling value co-creation in customer journeys with VIVA. In: Proceedings of Joint International Conference of Service Science and Innovation and Serviceology.

Sandkuhl, K., Stirna, J., Persson, A., Wißotzki, M., 2014. Enterprise Modeling: Tackling Business Challenges with the 4EM Method. Springer, Berlin.

Semeráth, O., Barta, Á., Horváth, Á., Szatmári, Z., Varró, D., 2017. Formal validation of domain-specific languages with derived features and well-formedness constraints. Softw. Syst. Model. 16 (2), 357–392.

Strembeck, M., Zdun, U., 2009. An approach for the systematic development of domain-specific languages. Softw. Pract. Exper. 39 (15), 1253–1292.

Tolvanen, J.-P., Kelly, S., 2009. MetaEdit+: Defining and using integrated domain-specific modeling languages. In: Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications. OOPSLA '09, Association for Computing Machinery, New York, NY, USA, pp. 819–820.

Torlak, E., Jackson, D., 2007. Kodkod: A relational model finder. In: Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007). In: LNCS 4424, pp. 632–647.

Wang, Y., Chen, Q., Hong, T., Kang, C., 2019. Review of smart meter data analytics: Applications, methodologies, and challenges. IEEE Trans. Smart Grid 10 (3), 3125–3148.

Weidmann, N., Kannan, S., Anjorin, A., 2021. Tolerance in model-driven engineering: A systematic literature review with model-driven tool support. CoRR abs/2106.01063.