# Towards Extending the Validation Possibilities of ADOxx with Alloy

Sybren De Kinderen, Qin Ma, Monika Kaczmarek-Hess

# Towards Extending the Validation Possibilities of ADOxx with Alloy

Sybren de Kinderen[1], Qin Ma[2], and Monika Kaczmarek-Heß[1]

[1] University of Duisburg-Essen, Universitätstrasse 9, D-45141 Essen, Germany
{sybren.dekinderen,monika.kaczmarek}@uni-due.de
[2] University of Luxembourg, 2 Avenue de l'Université, 4365 Esch-sur-Alzette, Luxembourg
qin.ma@uni.lu

**Abstract.** While ADOxx is a popular platform for the creation and use of enterprise modeling languages, it provides only limited support for a well-formedness check of created enterprise models. In this paper, we propose to complement the meta modeling platform ADOxx with Alloy, which natively provides extensive model checking capabilities, so as to enable a well-formedness check of enterprise models created in ADOxx. Using the $e^3value$ modeling language as a point of departure, we particularly provide (a) a partial ADOxx implementation of $e^3value$, (b) a proof-of-concept XML2Alloy parser, which allows for converting $e^3value$ models created in ADOxx into Alloy format, so that (c) $e^3value$ well-formedness constraints stated in Alloy can be used to check the validity of an $e^3value$ model with the Alloy Evaluator. Beyond the specific proof-of-concept, we also discuss further possibilities of using ADOxx in conjunction with Alloy, particularly in checking the soundness of meta models underlying an enterprise modeling language.

**Keywords:** meta modeling platforms · ADOxx · Alloy · Well-formedness

## 1 Introduction

Meta modeling platforms are an important stepping stone for the development, use, and dissemination of enterprise modeling methods [9, 10], by supporting the development of modeling environments, the creation of machine-readable models amenable to computer-supported analysis, offering querying capabilities, and more [9, 26, 10, 8].

There exist a wide variety of meta modeling platforms, with each having its own strengths and weaknesses, such as, e.g., Eclipse EMF [25], Visual Studio DSL Tools [6], MetaEdit+ [26], and ADOxx [9]. In this paper we focus on ADOxx. The reasons for this include that: it is also[3] widely used for implementing enterprise modeling (EM) languages, e.g., [3, 4, 24]; it is an openly available

[3] ADOxx has been used (1) by business to realize various products, e.g., ADONIS, (2) by researchers/academia, cf. the Open Models community at OMiLAB [5], as well as (3) in various European research projects, cf. [7].

modeling platform, which makes it easier to share development efforts [3]; and in our experience, at least for a basic implementation, it requires a minimum of coding for language development.

As we detail further in Section 2.1, ADOxx [9] provides a powerful means for the development and use of enterprise modeling languages. However at the same time, ADOxx provides only a limited means for ensuring well-formedness, both in terms of created enterprise models, as well as meta models underlying their creation. In particular, ADOxx natively lacks well-formedness check mechanisms, as a result requiring workarounds either in terms of (1) having to manually check query results, or (2) an automated check at the expense of a notable amount of additional syntax, e.g., event handlers, defining notifications, commands for query evaluation combined with escape characters to cope with quotes (") in the query, and more.

As a response, as also remarked by [18], it would be sensible to complement ADOxx with an instrument with dedicated model checking capabilities. Alloy [16] is a notable candidate in this regard. It is both a formal language for specifying complex structures, constraints, and model behavior, as well as an analyzer for automatically checking properties of models or simulating the execution of models.

The purpose of this paper is to show how Alloy complements ADOxx in terms of well-formedness checks of enterprise models. Using a partial implementation of the enterprise modeling language $e^3value$, we firstly show the workarounds required by ADOxx to implement well-formedness checks. Afterwards, we discuss how to leverage Alloy to perform well-formedness checks on models via an XML2Alloy parser. Here, we focus on the native model checking capabilities of Alloy, which allows one to declare in a concise manner any constraints and evaluate models against these. This is opposed to an automated check in ADOxx which, as stated, requires a considerable amount of additional syntax. Hence, this makes constraint checking with ADOxx time consuming and arguably error-prone.

This paper is a first step in a larger research project which aims at capitalizing on the complementary strengths of two respective meta modeling platforms: ADOxx, for creating and using enterprise models, and Alloy, for model checking capabilities. While in this paper we focus on showing the well-formedness check of models, of more general interest in our larger project are Alloy's capabilities to validate meta model specifications. Especially Alloy offers instance generation capabilities, which allows for the detection of abnormal instances, and subsequent adjustment of the meta model. We envision that, for implementing an enterprise modeling language for which formal validation is important – be it for a profitability analysis like with $e^3value$, or for a workload analysis on formalized enterprise architecture models like in [23, pp. 189-220] – such an integration allows for an attractive enterprise modeling language development workflow: on the one hand, to use the comprehensive features of ADOxx for enterprise modeling language development and use; on the other hand, to use Alloy for ensuring a consistent abstract syntax and formal semantics of the language.

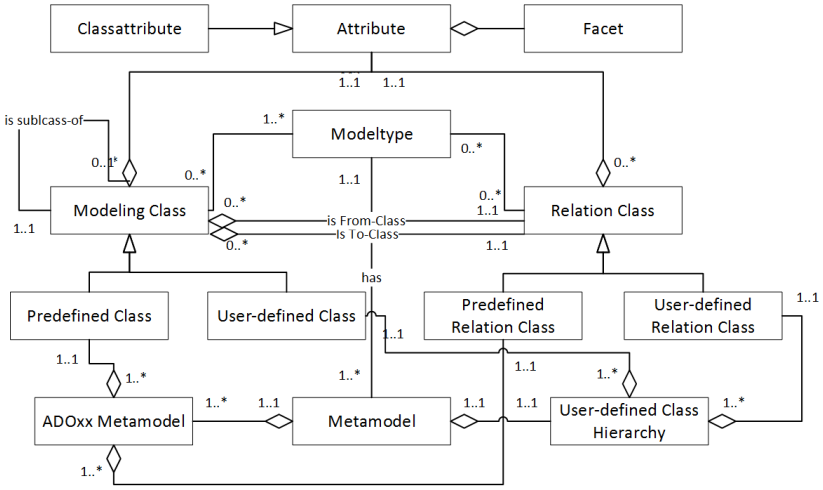



Fig. 1: An excerpt of the meta meta model as used by ADOxx Development Toolkit [9]

Please note that the idea for complementing ADOxx with additional tools/ platforms is not new. ADOxx has been coupled to GraphDB [21], to allow among others for semantic graph-based reasoning. Also, a relation of ADOxx to ConceptBase has been proposed [18], which, similar to our proposed integration with Alloy, allows for constraint checking of models created in ADOxx (for a further discussion, see Section 4 and Section 5).

This paper is structured as follows. In Section 2 we introduce ADOxx and Alloy, with a focus on their respective capabilities. Subsequently, using a partial implementation of the value modeling language *e*$^3$*value*, Section 3 shows the workarounds needed to check well-formedness in ADOxx, and shows, after transformation by means of an XML2Alloy parser, how Alloy can check the well-formedness of models created in ADOxx. Section 4 discusses related work, while Section 5 concludes the paper with final remarks.

## 2 Background: ADOxx and Alloy for EM tool development

### 2.1 ADOxx: DSML development and use

ADOxx consists of two environments for developing, using, and disseminating enterprise modeling languages: the ADOxx Development Toolkit, which is used to implement a language specification (at the $M_2$ level); and the ADOxx Modelling Toolkit, which is dedicated to modeling (at the $M_1$ level), and which provides further model analysis and export facilities [9, pp. 6—7].

In implementing a language specification with the ADOxx Development Toolkit, a language designer or developer needs to, for a given language, firstly

define the abstract syntax by mapping meta model elements to the concepts from the ADOxx meta meta model, cf. Figure 1. For one, classes and associations (in UML parlance) would in ADOxx be translated to classes, respectively relationclasses.[4] In addition by using ADOScript, ADOxx's scripting language [9, p. 15], the language designer can define a concrete syntax. Furthermore, the ADOxx development toolkit offers model types, which group classes and relationclasses for instantiation purposes. A language designer can capitalize on such grouping to, e.g., on the basis of one large language specification create different diagrams with different subsets of language elements. Finally, if of interest, the ADOxx Development Toolkit can be used to implement additional functionality, such as the possibility to pre-define queries to be executed on models created in the ADOxx Modeling Toolkit.

Subsequently, the ADOxx Modelling Toolkit is dedicated to language use. As such, it allows for creating models with the language specification defined in the ADOxx Development Toolkit. Additionally, the Modeling Toolkit also offers several possibilities of analysis on the created models. A prominent feature is the possibility to query models by means of the proprietary query language AQL. Also, the ADOxx Modeling Toolkit allows for an XML export of created models. This is especially important for interoperation with other tooling environments like, for this paper, Alloy.

As hinted by the above discussion, ADOxx provides a comprehensive variety of means for specifying and using enterprise modeling languages. However, as stated in the introduction, ADOxx is natively limited when it comes to model checking and simulation capabilities. Prominently of interest to this paper, this concerns the possibility to check for the well-formedness of enterprise models. As we concretely show in Section 3, ADOxx offers two possibilities: (1) queries, potentially pre-defined,[5] formulated using the query language AQL. Such queries may be used to check, among others, values of attributes, coherence of elements, the compliance with pre-defined rules and restrictions; (2) combining AQL queries with ADOScript. In this case, by combining ADOScript features, like, e.g., event handling and control structures, with querying capabilities, one can automate a well-formed check.

However, neither of the two possibilities is ideal. In the case of (pre-defined) queries, one has to check well-formedness partially *manually*, based upon the results of the queries. In addition, formulating consistency checking queries may be difficult, especially for inexperienced users. In the case of combining queries with ADOScript scripting functionalities, one has to introduce workarounds (like event handlers, or escape characters in queries, cf. also Section 3), which have little to do with the constraint checking per se. Thus, one essentially introduces accidental complexity, meaning added complexity due to the characteristics of

[4] Please note that UML does not fully match the ADOxx meta meta model. For example, UML roles do not have a clear counterpart in ADOxx. As such their implementation requires a workaround.

[5] AQL queries can be ad hoc formulated by a user, or pre-defined by a meta model developer in the Development Toolkit.

the chosen solution rather than characteristics of the underlying problem to be addressed [2]. This accidental complexity at least makes implementing an automated constraint check into ADOxx more time-consuming. Arguably, the added syntax also makes introducing constraints more error-prone.

As ADOxx offers the possibility to import and export models in a generic XML format, third-party components, e.g., external inference or reasoning tools, can be coupled with ADOxx to provide complementing functionalities [5]. As a response to the shortcomings of ADOxx mentioned above, we argue that Alloy, with its native support for model checking, provides a nice complement to ADOxx in terms of well-formedness checks.

### 2.2 Alloy: model checking and simulation

Alloy [16], is both a formal language for specifying complex structures, constraints, and behaviors of models, and an analyzer for automatically checking properties of models or simulating the execution of models. The Alloy specification language is based on first-order relational logic. An Alloy model specified in the Alloy language mainly consists of a set of *signatures*, declared with the keyword **sig**. Signatures in Alloy are similar to ADOxx classes. A signature may have zero or many *fields*, separated by comma. A field defines a relation between the instances of the containing signature and the instances of another signature.[6] 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; (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 [27]. 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" [1]. In other words, if there is an instance (or a counter example) of an Alloy model, there is one of small size.

Instances of an Alloy model can be further inspected for additional well-formedness constraints, by expressing them in terms of Alloy logical fomulae and evaluating the formulae on the instances using the functionality Alloy Evaluator, offered by the Alloy Analyzer [27]. Checking of well-formedness constraints on an instance in Alloy is fully automated. In contrast, the AQL based solution presented in Section 2.1 demands human intervention. Moreover, Alloy supports a full-fledged first-order relational logic, yet adopts an intuitive and succinct

[6] 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.

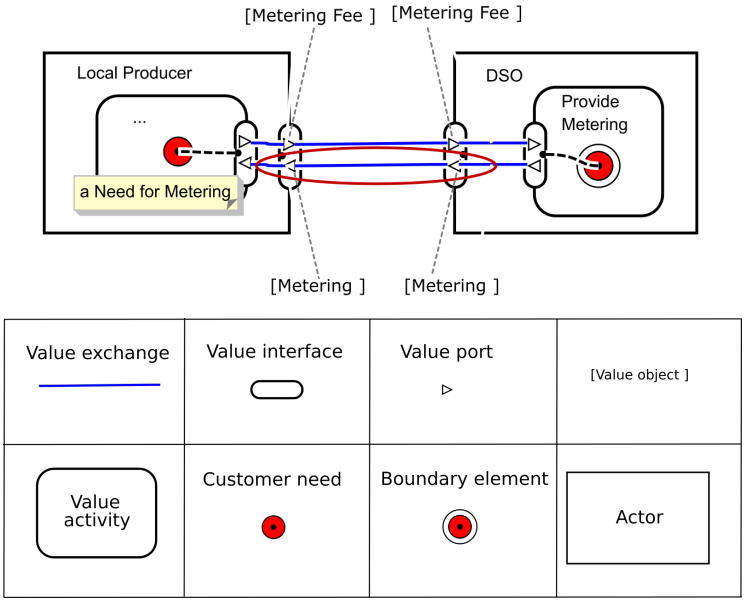



Fig. 2: An example $e^3value$ model

syntax for expressing formulae in the logic. Compared to the AQL and ADOScript solution offered by ADOxx (cf. Section 2.1), expressing well-formedness constraints in Alloy is much more simplified and efficient, as we will demonstrate with the $e^3value$ example in Section 3.

## 3 Case: Well-Formed Value Models

Now we show how specifically ADOxx can be used in conjunction with Alloy to allow for a well-formedness check of enterprise models. As an example, we focus on checking the well-formedness of models created with the enterprise modeling language $e^3value$, which comes equipped with a set of constraints.

### 3.1 Value modeling with $e^3value$

The value modeling language $e^3value$ [15] focuses on designing and analyzing value networks. Originally developed for supporting a business case analysis of e-business ideas, $e^3value$ is commonly used to provide answers to questions like: what are the actors involved in a value network? what do they provide of value and ask in return?

Figure 2 shows a simplified value model for a scenario in the electricity domain, which is loosely inspired by [14]. The scenario concerns a value exchange

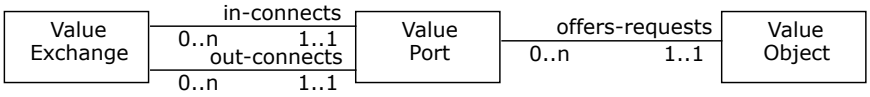



Fig. 3: A segment of the $e^3value$ formal ontology, implemented into ADOxx, cf [13, p.48]

between two actors: a Distributed Systems Operator (`DSO`), and a Local Producer of Electricity (`Local Producer`), whereby the `DSO` offers the value object `Metering` to the `Local Producer`, and receives in return the value object `Metering Fee`. Further, although less relevant for our purposes, the value exchange is triggered by the need: *a Need for Monitoring*, on the part of the `Local Producer` to produce the value object `Metering`, while the `DSO` executes the value activity `Provide Metering`.

In this paper, we focus on a partial implementation of $e^3value$ only. More specifically, we implemented a segment of the $e^3value$ formal ontology as shown in Figure 3. From this segment it can be observed that we focus on the meta types `Value Exchange`, `Value Port`, and `Value Object`, and the associations among them. For the electricity scenario presented above, an instantiation of this segment (highlighted in Figure 2 with a red circle) could show, how a given value exchange connects two value ports: one in, and one out, and that to each of the ports from the value exchange, there is a value object attached, namely `Metering`. Moreover, we include one well-formedness constraint relevant to the selected segment. In particular, this constraint, expressed in Listing 3.1, expresses 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. Returning to our electricity scenario, this constraint could for example check that, for our example value exchange, the same value object `Metering` is attached to both its in port and its out port.

Listing 3.1: The implemented constraint, which for a given value exchange ensures the consistency of the exchanged value object

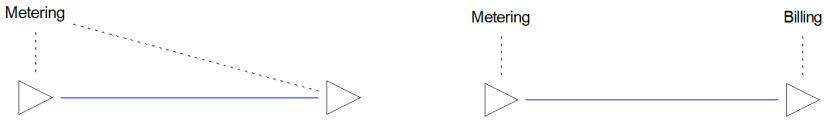
```
context value-exchange  inv:
        self.has-in.value-port.offers-requests.value-object =
        self.has-out.value-port.offers-requests.value-object
```

Note here that we provide only a partial implementation, since our case is meant as a proof-of-concept of the complementarity of ADOxx and Alloy. As such, a full implementation is beyond our scope.

### 3.2 Implementation of the $e^3$*value* segment in ADOxx

Using the ADOxx Modeling Toolkit, we implemented the partial language specification using the mechanisms briefly described in Section 2.1. In so doing we have defined `Value Exchange`, `Value Port`, and `Value Object` as classes in ADOxx. The associations and according cardinalities have been translated into relationclasses, each having the name of a role at its association end (so we have, e.g., the relationclass `in-connects`).[7] Also, for each model element so defined a concrete syntax has been assigned using ADOxx's scripting language, in line with the concrete syntax from $e^3$*value*. For example, the relationclasses `in-connects` and `out-connects` are visualized with GRAPHREP PEN w:.05 color:blue EDGE.

Figure 4 shows two example models created with the ADOxx modeling environment, instantiating the described partial $e^3$*value* implementation. Here the modeling environment natively assures conformance of the created models to the language specification in terms of (1) permissible relationclasses, e.g., a value port can be related to a value object via the `offers-requests` relation, cf. the language specification from Figure 3, and (2) cardinalities, e.g., a value port can be related to exactly one value object via the `offers-requests` relation.




(a) Well-formed (b) Not well-formed

Fig. 4: Partial $e^3$*value* models, created with ADOxx

### 3.3 Well-formedness check with ADOxx

However, importantly, as stated in Section 2 for ensuring well-formedness in terms of constraint checking workarounds in ADOxx are required. So when we want to check the compliance of the example models from Figure 4 to our constraint in Listing 3.1, concerning the two possibilities mentioned in Section 2 we either:

*1. (Pre-)define a AQL query, and manually check its result.* In this case, corresponding to the constraint in Listing 3.1 we define a query as follows:

Listing 3.2: Defined AQL Query

```
((<"E3_Value_Exchange_AC">->"out-connects")->"offers_requests") AND
((<"E3_Value_Exchange_AC">->"in-connects")->"offers_requests")
```

[7] As stated, as far as we know, there is no counterpart to UML roles in the ADOxx development environment.

This query returns, for all value exchanges whose in and out port have the same value object attached to it, the unique id-s, and (when specified) names, of the value object. This allows one to manually identify well-formed value exchanges;

*2. Automate the check.* Here, we combine queries formulated in AQL and the ADOScript scripting functionalities. In this case, an example implementation can look as follows:

Listing 3.3: Combined AQL query and ADOScript functionality

```
ON_EVENT "AppInitialized" {
CC "Application" INSERT_MENU_ITEM
    component:"modeling" item:"E3constraints"

CC "Application" INSERT_MENU_ITEM
    component:"modeling" item:"E3constraints\tConsistentObjects ..."
{
CC "Modeling" GET_ACT_MODEL

 CC "AQL" EVAL_AQL_EXPRESSION expr:
"((<\"E3_Value_Exchange_AC\">->\"out-connects\")->\"offers_requests\") AND
((<\"E3_Value_Exchange_AC\">->\"in-connects\")->\"offers_requests\")"

modelid: (modelid)
SETL c1: (objids)

CC "AQL" EVAL_AQL_EXPRESSION expr:
"((<\"E3_Value_Exchange_AC\">->\"out-connects\")->\"offers_requests\") OR
((<\"E3_Value_Exchange_AC\">->\"in-connects\")->\"offers_requests\")"

modelid: (modelid)
SETL c2: (objids)

SET c1c2diff: (tokdiff (c1,c2))

IF (c1c2diff != "") {

CC "AdoScript" INFOBOX ("The model is not well-formed. The value
exchanges with value objects " + c1 + "are valid, but the value
exchanges with value objects " + c2 + " are not." )
 }
 ELSE {
     CC "AdoScript" INFOBOX "The model is valid!"
 }
}
}
```

This example implementation inserts additional menu items in the ADOxx modeling environment (through `INSERT_MENU_ITEM`).
After a user triggers the inserted menu item, the open and active model is put in focus (through `GET_ACT_Model`), followed by `EVAL_AQL_EXPRESSION`, through which ADOScript signals that a query follows. Subsequently the same query follows, which in option 1 is evaluated manually, only now with escape characters added, to signal that a quote (") is used as part of the query (and so, it should not be parsed as being part of ADOScript). Then, after having explicitly stated that the query is indeed executed only on the active model (through `modelid`), the local variable `c1` is initialized with ids of the objects that result out of the query (through `objids`). The same logic is used to execute a second query, which for each value exchange returns all value objects offered through attached ports,

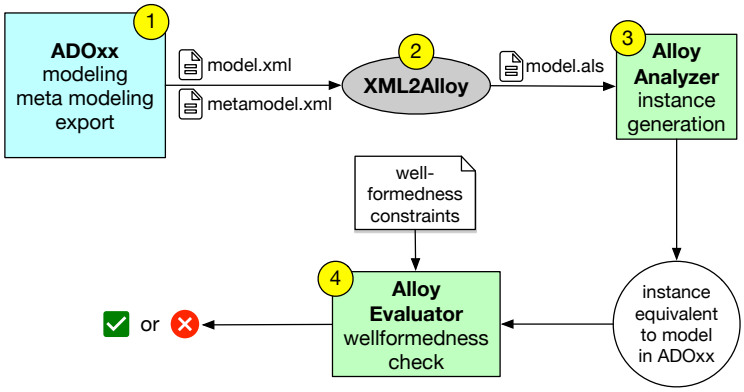



Fig. 5: Chaining ADOxx with Alloy for well-formedness checking

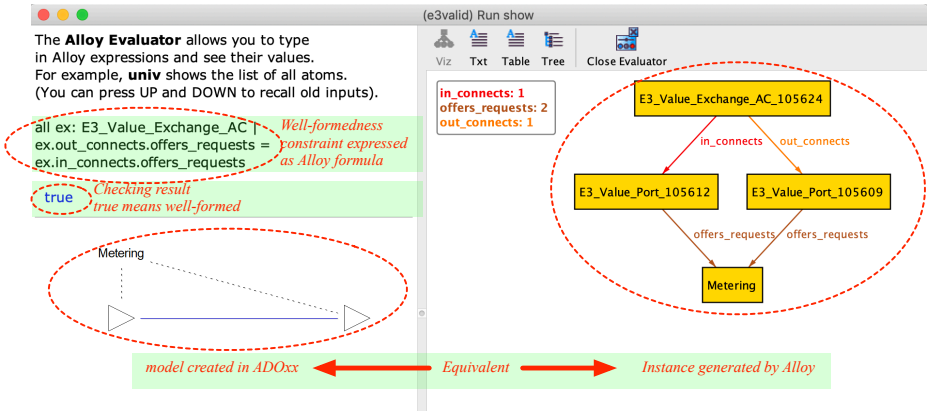
and to initialize the variable `c2` with the result. Afterwards in case the difference between the values of `c1` and `c2` is not empty (checked through `tokdiff (c1,c2)`), it means there exist value exchanges which have value objects that are not the same. Then, accordingly, a message will be displayed that helps the modeler identify the value exchanges with different value objects. Otherwise, a message will be displayed that the model is valid. This is of course one possibility of an automated check. Other possibilities can also be used, e.g., by using a different event handler one can execute the script every time that a model element is added, by using different queries, or by using different logical operators.

Nevertheless, what the example's implementation nicely illustrates, is the accidental complexity resulting out of the combination of ADOScript and AQL. For one, this concerns the way in which one has to announce that a query follows with the command `EVAL_AQL_EXPRESSION`, compounded by the fact that escape characters have to be inserted into the query any time a quote is used. Or, that to enable various automated checks, one has to initialize variables with very specific commands (like `objids`), with a subsequent extensive use of ADOScript control structures. In any case, the result is that one has to use a considerable amount of syntax to automate constraint checking in ADOxx. With Alloy, as a minimal declarative language, this can be expressed in a more concise and natural manner, as shown next.

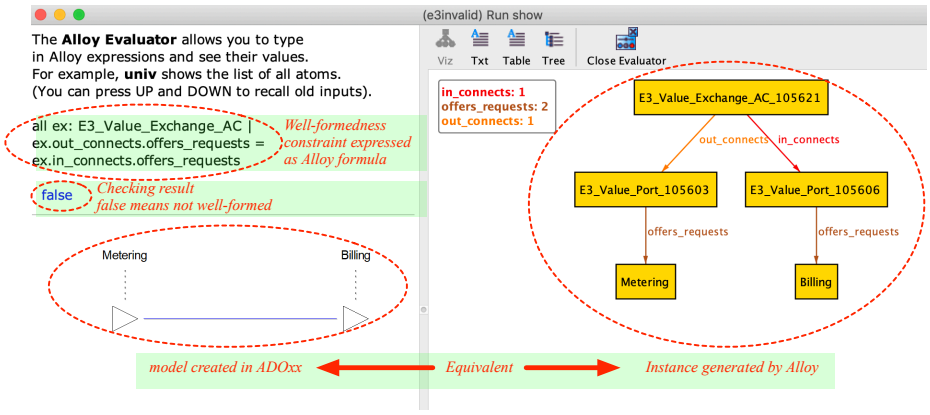
### 3.4 Well-formedness check with Alloy

In this section, we demonstrate how to leverage the power of Alloy to check the well-formedness constraints on models created in ADOxx. As depicted in Figure 5, such a check requires chaining ADOxx with Alloy. More specifically, firstly, a model created in ADOxx and the corresponding meta model developed in ADOxx are both exported in XML format (model.xml and metamodel.xml). Secondly, an XML2Alloy parser takes in these two XML files, and outputs an Alloy model (model.als). In this step, we capitalize on the "instance promotion"

strategy [11] to control the output Alloy model, so that when executed by the Alloy Analyzer, one and only one instance can be generated, which is equivalent to the model created in ADOxx. After generating the instance using Alloy Analyzer in the third step, fourthly, the well-formedness constraints are expressed in terms of Alloy logical fomulae, which are then evaluated on the instance (equivalent to the concerned ADOxx model) using the Alloy Evaluator functionality. All the links of this tool chain are already in place, provided either by ADOxx (for step 1), or by Alloy (for step 3 and step 4), except for step 2. As a proof of concept, we implement a prototype XML2Alloy parser in Python so as to make the chain complete.




(a) Model in Fig. 4a is well-formed.




(b) Model in Fig. 4b is not well-formed

Fig. 6: Well-formedness of partial $e^3value$ models from Figure 4, checked with Alloy

Figure 6 shows the final results of applying the tool chain to check the well-formedness constraint presented in Listing 3.1, on the two partial $e^3value$ models presented in Figure 4, respectively. For example, consider the model in Figure 4a. On the right hand side of Figure 6a, we see the instance equivalent to this model (visualized in a graphical representation of Alloy). Note that following the tool chain, this instance is generated by the Alloy Analyzer from the Alloy model that is produced by the XML2Alloy parser from the XML export of the model from ADOxx. More specifically, the Alloy model produced by the XML2Alloy parser from the model in Figure 4a is as follows:

Listing 3.4: Alloy model produced by the XML2Alloy parser

```
abstract sig E3_Value_Port {
     offers_requests: one Value_Object,
}{}

abstract sig Value_Object {
}{}

abstract sig E3_Value_Exchange_AC {
     in_connects: one E3_Value_Port,
     out_connects: one E3_Value_Port,
}{}

one sig E3_Value_Port_105609 extends E3_Value_Port{}{
     offers_requests = Metering
}

one sig E3_Value_Port_105612 extends E3_Value_Port{}{
     offers_requests = Metering
}

one sig E3_Value_Exchange_AC_105624 extends E3_Value_Exchange_AC{}{
     out_connects = E3_Value_Port_105609
     in_connects = E3_Value_Port_105612
}

one sig Metering extends Value_Object{}{}

pred show{}
run show
```

On the left hand side of Figure 6a, the well-formedness constraint is expressed as an Alloy formula, namely

Listing 3.5: Alloy formula expressing the well-formedness constraint

```
all ex: E3_Value_Exchange_AC |
ex.out_connects.offers_requests = ex.in_connects.offers_requests
```

The evaluation result is shown underneath it, where "true" means the instance satisfies the formula (namely the equivalent model is well-formed). In a similar manner, the well-formedness checking of the model in Fig. 4b is given in Figure 6b, and the result is "false". This means that the model is not well-formed.

## 4 Related Work

To extend ADOxx with validation capabilities, a few efforts exist. For instance, ADOxx has been used together with GraphDB [21] to capitalize on the graph-

based inference of the latter. The closest to our effort comes the work from the SemCheck project [18], which among others proposes a coupling of ADOxx with ConceptBase [17, 19] so as to enable constraint checking of, especially, multi-perspective enterprise models. The proposed integration architectures of both our work and SemCheck are similar. Both connect ADOxx with an external platform, i.e., Alloy or ConceptBase, via an intermediate, i.e., a XML2Alloy parser or ADOxx/Telos Adapter. This allows for translation of ADOxx (meta) models to the target platform, i.e., in terms of the Alloy syntax or the ConceptBase syntax.

The difference between the two approaches exhibits itself in the connected platforms. It is difficult to draw a judgement between the two as both have their strengths and weaknesses. For example, ConceptBase focuses more on internal model validation [18], i.e., to check if a (meta) model conforms to a (meta) meta model. It addresses less so external model validation, i.e., the check of dynamic semantics of (meta) models. In contrast, a mainstream use case scenario of Alloy is to validate the dynamic semantics of meta models by automatically generating instances of partial specifications. This allows for detecting abnormal instances and subsequently completing/correcting meta model definitions, e.g., cf. [12]. Moreover, Alloy supports model execution simulation and checking properties of such execution traces [22]. Differently, ConceptBase has been extended with axioms to support multi-level modeling [20].

In addition, different non-functional properties of the connected platforms may also be a factor in the acceptance of the two approaches. In this aspect, being intuitive to use, easy to learn, efficient to reason, popular in education, etc., are among the concerns one may want to consider. As a concrete example, although both ConceptBase and Alloy are based on relational 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.

## 5 Conclusions

In this paper, we propose to complement the meta modeling environment ADOxx with Alloy, so as to enable a well-formedness check of enterprise models created with ADOxx. As a proof-of-concept, we showed how $e^3value$ models created with a partial ADOxx implementation could be translated into an Alloy model, for subsequent well-formedness checks with the Alloy Analyzer.

For future work, as per the introduction, we intend to also capitalize upon the capability of Alloy to check the validation of meta models. Also for future work a more comprehensive comparison to the proposed integration with the SemCheck Conceptbase would be warranted, since the ideas standing behind that effort appear close to ours. Here, it would be particularly interesting to compare the meta model consistency checking capabilities of both environments. Especially, Conceptbase appears to check meta models against meta meta models [18, p. 41], whereas (as stated) Alloy pursues the discovery of inconsistencies

by generating instances, whereby observed abnormalities allow one to adjust the language specification of the enterprise modeling language at hand. Note here that one approach is not necessarily "better" than the other, it more depends on the support for use scenarios one is after. Finally, we intend to explore easing the integration between ADOxx and Alloy. Ideally speaking, we aim at realizing an on-the-fly well-formed check, supported by either developing an ADOxx plug-in (in the spirit of [21]), or by capitalizing on the promising capabilities of the Olive microservices framework, developed by OMiLAB to support modular meta modeling platform development and dissemination [5, pp. 683-684].

## References

1. Andoni, A., Daniliuc, D., Khurshid, S.: Evaluating the "small scope hypothesis". Tech. rep., MIT-LCS-TR-921, MIT CSAIL (2003)
2. Atkinson, C., Kühne, T.: Reducing accidental complexity in domain models. Software & Systems Modeling **7**(3), 345–359 (2008)
3. Bock, A., Frank, U.: Multi-perspective enterprise modeling – conceptual foundation and implementation with ado*xx*. In: Domain-Specific Conceptual Modeling, Concepts, Methods and Tools, pp. 241–267. Springer (2016)
4. Bork, D.: Metamodel-based analysis of domain-specific conceptual modeling methods. In: Buchmann, R.A., Karagiannis, D., Kirikova, M. (eds.) The Practice of Enterprise Modeling. pp. 172–187. Springer International Publishing, Cham (2018)
5. Bork, D., Buchmann, R., Karagiannis, D., Lee, M., Miron, E.T.: An Open Platform for Modeling Method Conceptualization: The OMiLAB Digital Ecosystem. Communications of the Association for Information Systems **44**, 673–697 (May 2019). https://doi.org/10.17705/1CAIS.04432, http://eprints.cs.univie.ac.at/5462/
6. Cook, S., Jones, G., Kent, S., Wills, A.C.: Domain-Specific Development with visual studio DSL tools. Pearson Education (2007)
7. Efendioglu, N., Woitsch, R.: Modelling method design: An adoxx realisation. In: 2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW). pp. 1–8. IEEE Computer Society, Los Alamitos, CA, USA (sep 2016). https://doi.org/10.1109/EDOCW.2016.7584376, https://doi.ieeecomputersociety.org/10.1109/EDOCW.2016.7584376
8. Englebert, V., Heymans, P.: Towards more extensible metacase tools. In: Krogstie, J., Opdahl, A., Sindre, G. (eds.) Advanced Information Systems Engineering. pp. 454–468. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
9. Fill, H., Karagiannis, D.: On the conceptualisation of modelling methods using the adoxx meta modelling platform. EMISA **8**(1), 4–25 (2013)
10. France, R.B., Frank, U., Oberweis, A., Rossi, M., Strecker, S.: Open models as a foundation of future enterprise systems (dagstuhl seminar 12131). In: Dagstuhl Reports. vol. 2. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2012)
11. Gammaitoni, L., Kelsen, P.: 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). pp. 324–327. LNCS 8477 (2014)
12. Gammaitoni, L., Kelsen, P., Glodt, C.: Designing languages using lightning. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2015). p. 77–82 (2015)
13. Gordijn, J.: Value-based requirements Engineering: Exploring innovative e-commerce ideas. Ph.D. thesis, Vrije Universiteit Amsterdam (2002)

14. Gordijn, J., Akkermans, H.: Business models for distributed energy resources in a liberalized market environment. The Electric Power Systems Research Journal **77**(9), 1178–1188 (2005), http://docs.e3value.com/bibtex/pdf/Gordijn2005DER.pdf, preprint available. doi:10.1016/j.epsr.2006.08.008
15. Gordijn, J., Akkermans, J.: e3-value: Design and evaluation of e-business models. IEEE Intelligent Systems pp. 11–17 (2001)
16. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2 2012), revised edition
17. Jarke, M., Gallersdörfer, R., Jeusfeld, M.A., Staudt, M., Eherer, S.: ConceptBase—a Deductive Object Base for Meta Data Management. J. Intell. Inf. Syst. **4**(2), 167–192 (Mar 1995). https://doi.org/10.1007/BF00961873, https://doi.org/10.1007/BF00961873
18. Jeusfeld, M.A.: SemCheck: Checking Constraints for Multi-perspective Modeling Languages, pp. 31–53. Springer International Publishing, Cham (2016)
19. Jeusfeld, M.A., Jarke, M., Nissen, H.W., Staudt, M.: Conceptbase: Managing conceptual models about information systems. In: Bernus, P., Mertins, K., Schmidt, G. (eds.) Handbook on Architectures of Information Systems, pp. 273–294. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
20. Jeusfeld, M.A., Neumayr, B.: Deeptelos: Multi-level modeling with most general instances. In: International Conference on Conceptual Modeling. pp. 198–211. Springer (2016)
21. Karagiannis, D., Buchmann, R.A.: A proposal for deploying hybrid knowledge bases: the ADOxx-to-GraphDB interoperability case. In: Proceedings of the 51st Hawaii International Conference on System Sciences (2018)
22. Kelsen, P., Ma, Q.: 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). pp. 690–704. LNCS 5301, Springer (2008)
23. Lankhorst, M.: Enterprise Architecture at Work: modeling, Communication and Analysis. Springer, 4 edn. (2017)
24. Schwab, M., Karagiannis, D., Bergmayr, A.: i* on adoxx®: A case study. In: iStar 2010–Proceedings of the 4 th International i* Workshop. p. 92 (2010)
25. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: eclipse modeling framework. Pearson Education (2008)
26. Tolvanen, J.P., Kelly, S.: 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. pp. 819–820 (2009)
27. Torlak, E., Jackson, D.: 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). pp. 632–647. LNCS 4424 (2007)