ECAVI: A tool for Event Calculus Analysis and Visualization

Parthena Basina

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Prof. *Dimitris Plexousakis*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

University of Crete
Computer Science Department

ECAVI: A tool for Event Calculus Analysis and Visualization

Thesis submitted by

Parthena Basina

in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author:	Parthena Basina
Committee approvals:	Dimitris Plexousakis Professor, Thesis Supervisor
	George Papagiannakis Assistant Professor, Committee Member
	Giorgos Flouris Principal Researcher, Committee Member
Departmental approval:	Antonios Argyros Professor, Director of Graduate Studies

Heraklion, March 2019

ECAVI: A tool for Event Calculus Analysis and Visualization

Abstract

Although action languages are well-established as a means to model dynamic domains, their adoption by knowledge engineers is often hindered by modelling errors and steep learning curves. Event Calculus (EC), as one of the most prominent action languages, has a way of representing causal and narrative information which differentiates from other similar formalisms. It has been argued that visual modelling tools could assist knowledge engineers in the modelling task and improve the quality of the resulting models. The ADOxx Meta-Modelling platform enables the development of modelling toolkits where the metamodel and the modelling method are made by the developer.

In this thesis, we present the tool ECAVI (Event Calculus Analysis and VIsualisation), a domain independent visual modelling tool for designing dynamic domains in the Event Calculus. ECAVI is mainly addressed to inexperienced modellers (such as students who are working towards understanding the Event Calculus), aiming to help them become acquainted with the features of Event Calculus and to guide them during the process of designing their problems.

For the realisation of the tool we make use of the ADOxx meta-modelling platform's functionalities in order to design our graphical language based on the syntax and semantics of the Answer Set Programming (ASP) formal language and with the help of a Java program, we pair it with the state-of-the-art automated reasoner, Clingo.

Even though ECAVI is still a work-in-progress, with several features that have been planned but not implemented yet, we argue that the tool will be useful to a diverse audience of knowledge modellers as a teaching assistant for the fundamental concepts of reasoning about actions and change and also as a way to visualise full ASP programs.

ECAVI: Ένα εργαλείο για Ανάλυση και Οπτικοποίηση του Λογισμού Συμβάντων

Περίληψη

Οι γλώσσες δράσης έχουν καθιερωθεί ως μέσο για τη μοντελοποίηση δυναμικών τομέων, ωστόσο, η υιοθέτησή τους από μηχανικούς γνώσης συχνά εμποδίζεται από σφάλματα μοντελοποίησης και απότομες καμπύλες μάθησης. Ο Λογισμός Συμβάντων (ΛΣ), ως μία από τις πιο σημαντικές γλώσσες δράσης, μπορεί να αναπαραστήσει αιτιώδεις και αφηγηματικές πληροφορίες με τρόπο που διαφοροποιείται από άλλους παρόμοιους φορμαλισμούς. Έχει υποστηριχθεί ότι τα εργαλεία οπτικής μοντελοποίησης μπορούν να βοιθήσουν τους μηχανικούς γνώσης κατά τη διαδικασία της μοντελοποίησης και να βελτιώσουν την ποιότητα των μοντέλων που προκύπτουν. Η Meta-Modelling πλατφόρμα ADOxx επιτρέπει την ανάπτυξη εργαλείων μοντελοποίησης όπου το μεταμοντέλο και η μέθοδος μοντελοποποίησης κατασκευάζονται από τον προγραμματιστή.

Σε αυτή τη διπλωματική εργασία, παρουσιάζουμε το εργαλείο ECAVI, ένα ανεξάρτητο τομέων εργαλείο οπτικής μοντελοποίησης για το σχεδιασμό δυναμικών τομέων στον Λογισμό Συμβάντων. Το ECAVI απευθύνεται κυρίως σε άτομα δίχως εμπειρία στη μοντελοποίηση (όπως μαθητές που βρίσκονται στο στάδιο της κατανόησης του ΛΣ) με στόχο να τους βοηθήσει να εξοικειωθούν με τα χαρακτηριστικά του ΛΣ και να τους καθοδηγήσει κατά τη διαδικασία σχεδιασμού των προβλημάτων τους.

Για την υλοποίηση του εργαλείου χρησιμοποιούμε τις λειτουργίες της metamodelling πλατφόρμας ADOxx ώστε να σχεδιάσουμε τη γραφική μας γλώσσα με βάση το συντακτικό και τη σημασιολογία της Answer Set Programming (ASP) γλώσσας, και με τη βοήθεια ενός Java προγράμματος τη συνδυάζουμε με τον σύγχρονο αυτοματοποιημένο reasoner, Clingo.

Παρόλο που το ECAVI αποτελεί ακόμα δουλειά σε εξέλιξη, με αρκετά χαρακτηριστικά που έχουν προγραμματιστεί αλλά δεν έχουν ακόμη εφαρμοστεί, υποστηρίζουμε ότι το εργαλείο θα είναι χρήσιμο σε ένα κοινό ποικίλων ατόμων που ασχολούνται με μοντελοποίηση γνώσης ως ένας βοηθός διδασκαλίας για τις θεμελιώδεις έννοιες της συλλογιστικής σχετικά με τις ενέργειες και την αλλαγή μέσα στο χρόνο αλλά και ως έναν τρόπο οπτικοποίησης πλήρων προγραμμάτων σε ASP.

Acknowledgements

First of all, I would like to thank my supervisor Professor Dimitris Plexousakis for giving me the opportunity to complete my master thesis on a topic that I find very interesting.

I would also like to express my deep gratitude towards my co-supervisors, Postdoctoral Researcher Theodore Patkos and Principal Researcher Giorgos Flouris of the Institute of Computer Science, ICS-FORTH, for providing me with guidance and support throughout the study of my master, for the precious advice they gave me in all the times I faced problems and for always being available when I needed them.

A special thanks to the Information Systems Laboratory (ISL) in the Institute of Computer Science (ICS) of the Foundation of Research and Technology Hellas (FORTH) for providing me with financial assistance because otherwise I wouldn't have made it. A big thank you to Ms. Maria Moutsaki, secretariat of ISL for all her help and support.

Furthermore, I'm also very grateful for my friends, for supporting me throughout all the hard times I experienced. Both the friends I made in the Computer Science Department since my first years as an undergraduate student have been there for me all these years, as well as, the friends I made in most recent years have helped me throughout many hardships. Eva, Giorgo, Michali, Aggele, Ioanna, Giorgo, Dimitri, Vangeli, Lydia, Alex, Fani, Panagioti, thank you! Without your help and understanding I wouldn't have made it this far.

And last but definitely not least, a massive thank you to my family, most specially to my parents Eleni and Prodromos and my sister Dimitra, for being there and supporting me in my decisions, even when they didn't agree with them. Their love and encouragement helped me overcome any difficulty I faced.

"The world ain't all sunshine and rainbows. It's a very mean and nasty place and I don't care how tough you are it will beat you to your knees and keep you there permanently if you let it.
You, me, or nobody is gonna hit as hard as life. But it ain't about how hard ya hit. It's about how hard you can get hit and keep moving forward. How much you can take and keep moving forward.
That's how winning is done! Now if you know what you're worth then go out and get what you're worth. But ya gotta be willing to take the hits, and not pointing fingers saying you ain't where you wanna be because of him, or her, or anybody!
Cowards do that and that ain't you! You're better than that!"

Rocky Balboa

"We must accept finite disappointment but never lose infinite hope!"

Martin Luther King

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Reasoning about actions, change and causality has been an important challenge from the early days of Artificial Intelligence (AI). Action languages are well-established logical theories for reasoning about the dynamics of changing worlds, aiming at "*formally characterizing the relationship between the knowledge, the perception and the action of autonomous agents*" [31]. One of the most prominent action languages is the Event Calculus [19, 25], which incorporates certain useful features for representing causal and narrative information and has been applied in domains as diverse as high-level robot cognition, argumentation [1, 5], service composition [30], complex event detection [32], and others.

The Event Calculus, being a logical formalism, is generally hard to tackle by the non-expert, and novice practitioners find it hard to properly model a domain of interest. It has been argued [26] that visualisations generally help knowledge engineers understand better the ramifications of their modelling decisions. In the context of the Event Calculus, we argue that a visual representation of the various axiom types may help knowledge engineers understand the semantics of the different axiom types, thereby simplifying the learning process for inexperienced modellers and reducing the number of modelling mistakes.

Fill and Karagiannis [8], investigated the role of visualization in the conceptualization of modelling methods and commented on the fact that "*the absence of a graphical representation during modelling will inevitably force the engineer to develop an adequate visualization for the elements of the syntax of the modelling language, by taking into account the corresponding semantics*". This realization is also enforced by many popular efforts in visualisation, such as the introduction of UML [9] in the context of software engineering, or Protégé[1] as a visual tool for ontology editing.

A model represents a partial and simplified view of a system, so, the creation of multiple models is usually necessary to better represent and understand the

[1]`https://protege.stanford.edu/`

system under study. Models allow the sharing of a common vision and knowledge among technical and non-technical stakeholders, facilitating and promoting the communication among them. They also make the project planning more effective and efficient while providing a more appropriate view of the system to be developed and allowing the project control to be achieved according to objective criteria. The concepts of system, model, meta-model and their relations are the essential concepts of Model-Driven Engineering (MDE) [3]. MDE focuses on the models, rather than the code, using them as primary engineering artefacts [2]. ECAVI takes a model-based approach, where the code is generated directly from the models, in order to meet integration and interoperability requirements in the context of MDE.

To the best of our knowledge, there does not exist any tool that focuses on the visualization of Event Calculus semantics. Visic et. al. [33] introduced the only good point-of-reference for our case, a domain-specific language (DSL) that considers the "modelling method engineering" as the application domain and allows the method engineer to focus on the conceptual building blocks of a modeling method rather than on a meta-modelling platform's technical specificity.

Motivated by this fact, the object of this thesis is the design and development of a new, domain independent modelling tool, ECAVI (Event Calculus Analysis and VIsualisation), which offers a visual language for designing dynamic domains in the Event Calculus, while assisting the user in the process of knowledge engineering, through the ADOxx meta-modelling platform and with the help of a state-of-the-art automated reasoner, Clingo.

1.2 Contributions

For the first version of the ECAVI modelling tool, we mainly focus on novice users of Event Calculus, with basic or no knowledge of the formalism, such as students that are working towards understanding the Event Calculus; our aim is for this tool to be used as an assistant for teaching the fundamental concepts of reasoning about actions and change and in the future, also assist the more experienced users. More specifically, the first version of the ECAVI modelling tool relies on the following contributions:

- The offer of a visual language for designing causal dynamic domains, supporting phenomena such as context-dependent event occurrences, context-dependent effects of events and concurrency. Next versions will further extend the language with more features, such as non-determinism, indirect effects of events and others.
- The tight coupling of the visual domain representation with two powerful logical formalisms, namely the Event Calculus and Answer Set Programming (ASP), that enable the knowledge engineer to perform complex reasoning tasks, such as progression, observation explanation etc.

- Assist the user in the process of knowledge engineering, minimising the possibility for syntactical errors. More importantly, our current work concerns also the raising of warnings and exceptions whenever logical fallacies are detected, which may lead to contradicting or counter-intuitive behaviour, e.g., when the same property may become true and false at the same time.
- Adoption of a pedagogical approach in the process of designing causal domains, aiming to help non-experts, such as students, to learn the basics of how a conceptual model can be translated and executed through the logic programming paradigm.

1.3 Thesis Outline

The thesis is organized as follows. Chapter 2 describes all the necessary background material needed for understanding the fundamentals of the 2 main parts our tool comprises of, namely Event Calculus and the ADOxx Meta-modelling tool. On Chapter 3 we review some tools that are relevant to our work and have provided valuable insight towards the realisation of ECAVI. Chapter 4 describes the full methodology behind the conceptualization of our tool and the visual notation adapted on the ASP syntax and semantics. Chapter 5 presents the main architecture of ECAVI, as well as, a full description of a simple use case implemented with the tool. Finally, chapter 6 presents the main conclusions of this thesis along with a description of the open issues and directions that define our future work.

Chapter 2

Background

This chapter provides the necessary background material in order to understand and follow the main concepts of this thesis. It comprises of 3 main parts that define our tool and are essential to our methodology and implementation: Event Calculus, Answer Set Programming (ASP) and the ADOxx Metamodelling Platform.

2.1 Event Calculus

Commonsense reasoning is essential to intelligent behavior and thought. It allows us to fill in the blanks, to reconstruct missing portions of a scenario, to figure out what happened, and to predict what might happen next. Reasoning about the world requires a large amount of knowledge about the world and the ability to use that knowledge. Commonsense reasoning can be used to make computers more human-aware, easier to use, and more flexible. Although it comes to us naturally and appears to be simple, it is actually a complex process [27].

The Event Calculus is a narrative-based many-sorted first-order language for reasoning about action and change. The basic notions of the Event Calculus are events, fluents and timepoints. It explicitly represents temporal knowledge, enabling reasoning about the effects of a narrative of events along a time line. It also relies on a non-monotonic treatment of events, in the sense that by default there are no unexpected effects or event occurrences.

Several fundamental entities must be represented: objects in the world and agents such as people and animals, properties in the world that change over time which we call fluents and such is the location of an object, events or actions that occur in the world such as the action of a person moving an object, and at last we need to represent time.

Formally, a sort $\mathcal{E}$ of *events* indicates changes in the environment, a sort $\mathcal{F}$ of *fluents* denotes time-varying properties and a sort $\mathcal{T}$ of *timepoints* is used to implement a linear time structure. The calculus applies the *principle of inertia* for fluents, in order to solve the frame problem, which captures the property that things tend to persist over time unless affected by some event.

An event may occur or happen at a timepoint and a fluent has truth value at a timepoint or over a timepoint interval (true or false). The occurence of an event may affect the state of a fluent. We have commonsense knowledge about the effects of events on fluents, specifically about, events that initiate fluents and events that terminate fluents. For example, we know that the event of picking up an object initiates the fluent of holding the object and the event of setting down an object terminated the fluent of holding the object. We represent these notions in first-order logic with the use of the Event Calculus domain-independent predicates[1]:

- *HoldsAt(F,T)* represents that fluent F is true at timepoint T.
- *Happens(E,T)* represents that event E occurs at timepoint T.
- *Initiates(E,F,T)* represents that, if event E occurs at timepoint T, then the fluent F will be true after T.
- *Terminates(E,F,T)* represents that, if event E occurs at timepoint T, then the fluent F will be false after `T`.
- *ReleasedAt(F,T)* represents that fluent F is released from the commonsense law of inertia at timepoint T and its truth value can fluctuate.

The commonsense notions of persistence and causality are captured in a set of *domain independent* axioms, referred to as $\mathcal{DEC}$ [27], that express the influence of events on fluents and the enforcement of inertia for the *holdsAt* and *releasedAt* predicates. In brief, $\mathcal{DEC}$ (Discrete time Event Calculus) states that a fluent that is not released from inertia has a particular truth value at a particular time if at the previous timepoint either it was given a cause to take that value or it already had that value. In $\mathcal{DEC}$ timepoints are restricted to the integers. For example, $initiates(e, f, t)$ means that if action e happens at timepoint t it gives cause for fluent f to be true at timepoint $t + 1$.

In addition to domain independent axioms, a particular domain axiomatisation requires also axioms that describe the commonsense domain of interest, observations of world properties at various times and a narrative of known world events. The role of the ECAVI tool is to assist knowledge engineers in designing Event Calculus domain axiomatisations, without requiring them to master the complexities of logic programming.

Satisfiability and logic programming-based implementations of Event Calculus dialects have been proposed over the years. Recently, progress in generalising the definition of stable model semantics [7] used in ASP has opened the way for the reformulation of Event Calculus axiomatisations into logic programs that can be executed with ASP solvers [21]. ASP is a form of knowledge representation and reasoning paradigm oriented towards solving complex combinatorial search

[1]In the sequel, variables, starting with a upper-case letter, are implicitly universally quantified, unless otherwise stated. Predicates and constants start with a lower-case letter.

problems. A domain is represented as a set of logical rules, whose models, called answer sets, correspond to solutions to a reasoning task, such as progression or planning. As will be described next, ECAVI implements a translation of Event Calculus theories into ASP rules, which are then executed by the Clingo ASP reasoner[2].

2.2 Answer Set Programming

Answer Set Programming (ASP) is an approach to knowledge representation and reasoning. Knowledge is represented as *answer sets programs*, and reasoning is performed by *answer set solvers.* Answer set programming enables default reasoning, which is required in commonsense reasoning.

The syntax of answer set programs derives from the Prolog language. We use the syntax of the ASP-Core-2 standard. The semantics of answer set programs is defined by the stable model semantics introduced by Michael Gelfond and Vladimir Lifschitz [11, 23], where a conclusion is infered only if there is explicit evidence to support it.

An answer set program consists of a set of rules of the form:

$$\alpha : -\beta.$$

which represents that α, the head of the rule, is true if β, the body of the rule, is true. Here is an answer set program:

```
p.
r :- p, not q.
```

The first rule `p.` is called a *fact.* It has an empty body and is written without the `:-` (if) connective.The symbol `,` indicates conjunction ($\wedge$). The token `not` refers to negation as failure and is different from classical negation ($\neg$). The expression `not q` represents that q is not found to be true.

We can perform automated reasoning on this program by placing it in a file `example.lp` and running the answer set grounder and solver `clingo` on the file. The clingo reasoner is a combination of the answer set grounder `gringo` and the answer set solver `clasp`, offering more control over the grounding and solving processes.

The syntax of answer set programs is defined as follows:

A ***signature*** consists of the following disjoint sets:

- A set of **constants**.
- For every $n \in \{1, 2, 3, ...\}$, a set of n-ary function symbols.
- For every $n \in \{0, 1, 2, ...\}$, a set of n-ary predicate symbols.

[2]https://potassco.org/

Given a signature σ and a set of variables disjoint from the signature, we define answer set programs as follows.

A ***term*** is defined inductively as:

- A constant is a term.
- A variable is a term.
- If τ_1 and τ_2 are terms, then $-\tau$, $\tau_1 + \tau_2$, $\tau_1 - \tau_2$, $\tau_1 * \tau_2$ and τ_1/τ_2 are terms. The symbols +,-,*, and / are *arithmetic* symbols.
- If ϕ is an n-ary function symbol and $\tau_1, ..., \tau_n$ are terms, then $\phi(\tau_1, ..., \tau_n)$ is a term.
- Nothing else is a term.

A ***ground term*** is a term containing no variables and no arithmetic symbols.
An ***atom*** is defined inductively as follows:

- If ρ is an n-ary predicate symbol and $\tau_1, .., \tau_n$ are terms, then $\rho(\tau_1, ..., \tau_n)$ is an atom.
- If ρ is an 0-ary predicate symbol, then ρ is an atom.
- If τ_1 and τ_2 are terms, then $\tau_1 < \tau_2$, $\tau_1 <= \tau_2$, $\tau_1 = \tau_2$, $\tau_1 ! = \tau_2$, $\tau_1 > \tau_2$, and $\tau_1 >= \tau_2$ are atoms. The symbols $<, <=, =, ! =, >$, and $>=$ are the *comparative predicates.*
- Nothing else is an atom.

A ***ground atom*** is an atom containing no variables, no arithmetic symbols, and no comparative predicates.
A ***rule*** is

$$\alpha_1 \mid ... \mid \alpha_k : -\beta_1, ..., \beta_m, \text{ not } \gamma_1, ..., \text{ not } \gamma_n.$$

where $\alpha_1, ..., \alpha_k$, $\beta_1, ..., \beta_m$, $\gamma_1, ..., \gamma_n$ are atoms. $\alpha_1 | ... | \alpha_k$ is the head of the rule, and $\beta_1, ..., \beta_m$, not $\gamma_1, ...,$ not γ_n is its body.
A ***fact*** is a rule whose body is empty (m = 0 and n = 0).
A ***constraint*** is a rule whose head is empty (k = 0).
A ***ground rule*** is a rule containing no variables, no arithmetic symbols, and no comparative predicates.
A ***logic program***, ***answer set program*** or ***program*** is a set of rules.
A ***traditional rule*** is a rule whose head contains a single atom (k = 1).
A ***traditional program*** is a set of traditional rules.
A ***ground program*** is a program containing no variables, no arithmetic symbols, and no comparative predicates.

Answer set programming languages, such as those of lparse, gringo, and DLV, and the standard ASP-Core-2, further specify the following:

- Constants are integers or start with a lowercase letter.
- Variables start with an uppercase letter.
- Function symbols start with a lowercase letter.
- Predicate symbols start with a lowercase letter.

2.3 Agile Modelling Method Engineering

The Agile Modelling Method Engineering (AMME) is a domain-independent methodology addressing the interaction between modelling and machine processing of models, including, simulation, analysis and code-generation. The main characteristics of AMME with regard to changing requirements are [12, 14]:

- **Adaptability**: the ability to modify existing concepts/properties (to meet new requirements).
- **Extensibility**: the ability to add new concepts/properties to the existing metamodel.
- **Integrability**: the ability to add bridging concepts/properties in order to integrate existing building blocks.
- **Operability**: the ability to provide satisfying user interaction and model understandability.

AMME relies on a methodological core called the Conceptualization Lifecycle which establishes several phases for incrementally deriving modelling tools, from modelling method creation until the technical deployment in the form of usable software [14]. Thus, in ECAVI, we focus on this methodology in order to define the requirements and design the modelling language of our tool.

2.4 ADOxx Metamodelling Platform

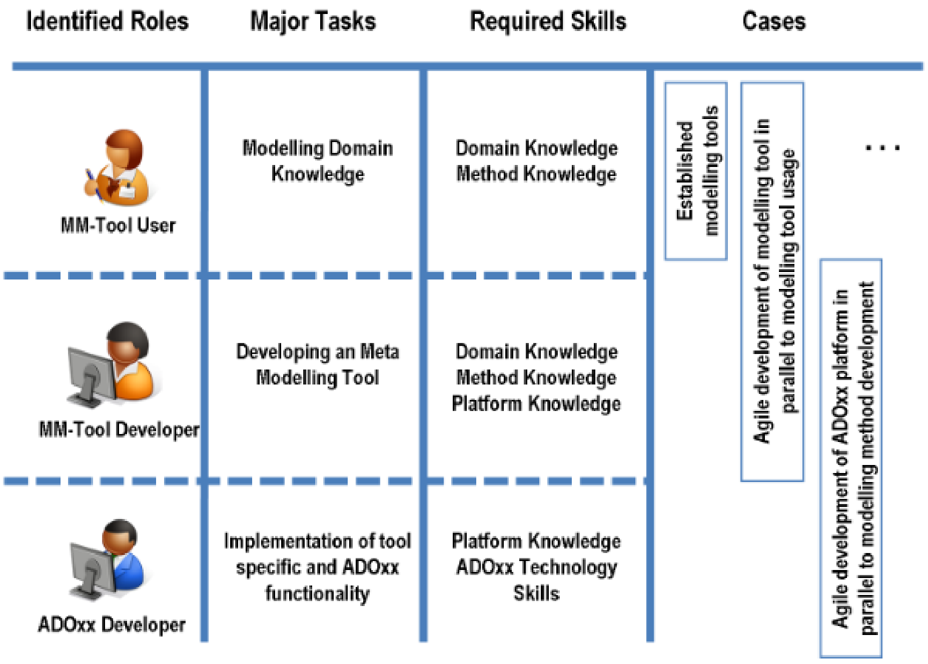
ADOxx is the creation of the Open Models Laboratory (OMiLAB)[3], a dedicated research and experimentation space for modelling method engineering. Both a physical and virtual place, it is equipped with tools to explore method creation and design, experiment with method engineering and deploy software tools for modelling [12].

The ADOxx meta-modelling development and configuration platform[4] enables the development of modelling toolkits, where the metamodel and the modelling method are made by the developer. The modelling toolkits implemented with ADOxx follow a configuration approach on platform level (re-use of existing implementations and functionality on platform level in different scenarios) that is supported by an expert community.

[3]`http://austria.omilab.org/psm/home`
[4]`https://www.adoxx.org/live/adoxx-documentation`

The tool has been used and tested for more than 20 years in research and industrial projects and is considered a mature tool for metamodel development with a great variety of features, high scalability and reliability [8].

In ADOxx the following roles are distinguished in relation to the task/skills needed to perform certain modelling/meta-modelling tasks.



Identified Roles	Major Tasks	Required Skills	Cases
MM-Tool User	Modelling Domain Knowledge	Domain Knowledge Method Knowledge	Established modelling tools; Agile development of modelling tool in parallel to modelling tool usage; . . .
MM-Tool Developer	Developing an Meta Modelling Tool	Domain Knowledge Method Knowledge Platform Knowledge	Agile development of modelling tool in parallel to modelling tool usage; Agile development of ADOxx platform in parallel to modelling method development
ADOxx Developer	Implementation of tool specific and ADOxx functionality	Platform Knowledge ADOxx Technology Skills	Agile development of ADOxx platform in parallel to modelling method development

Table 2.1: ADOxx Role/Task/Skill Matrix

- **Modelling Method Tool User:** represents the target group of the toolkit to be developed. The major task of this role relates to transforming the domain knowledge into requirements for the modelling toolkit.
- **Modelling Method Developer:** has expertise in developing and translating the requirements into meta-model concepts (class hierarchy, relation specification, cardinalities)
- **ADOxx Developer:** uses the input of the MM-Tool Developer and maps these requirements to available ADOxx functionality and implements the requirements accordingly.

In ECAVI, we have both the roles of the ADOxx Developer and the MM-Tool Developer in order to implement the modelling language that translates the requirements defined by the Tool User. Furthermore, ECAVI is designed so that any MM-Tool User with no method knowledge can gain this knowledge with the use of our tool.

ADOxx provides two different toolkits, both of them implementing the ADOxx meta^2 model and operating on the same database. The *Development Toolkit* supports the creation of modelling methods, whereas the *Modelling Toolkit* allows for the creation of models.

The specification and definition of a modelling method is defined by Karagiannis and Kühn [16]. Modelling methods are divided into two components: a *modelling technique*, consisting of the modelling language and the modelling procedure, and the *mechanisms and algorithms*. The *modelling language* contains the elements that describe the models and is defined by its syntax, semantics and notation. The *modelling procedure* is implicitly realised with the model types, whereas each one is a set of modelling concepts that are grouped in a useful way.

It is possible to define simple metamodels, which are the prerequisite of deploying modelling toolkits, based on three concepts and their relations. A *class* is one of the core constituents of the meta^2 model of ADOxx and can be related with other classes by means of *relation classes*, where it is possible to specify which class may be connected with which classes by means of the specific relation class [8, 17].

2.4.1 The GraphRep class attribute

The GraphRep class attribute[5] allows the design of a graphical representation for a specific constructs in a graphical design application. Upon completion, the graphical construct representation can be automatically translated into the platform-specific code of ADOxx. It is of type LONGSTRING, hence it's value is a text that is interpreted as a script by the GRAPHREP interpreter.

There are five types of elements distinguished: *Style elements*, *Shape elements*, *Variable assigning elements*, *Context elements* and *Control elements.*

The representation characteristic for following shape elements is modified by five **style elements**: PEN, FILL, SHADOW, STRETCH and FONT.

The ADOxx GraphRep repository collects implementation of graphical representation from different scenarios and projects and provides them to the community. All community members as free to add, revise, use, modify, comment and rate the GraphReps available in the repository[6].

[5]`https://www.adoxx.org/live/graphrep`

[6]`https://www.adoxx.org/live/adoxx-graphrep-repository-wiki/-/wiki/GRAPHREP+Repository/FrontPage`

2.4.2 AdoScript

ADOxx features a powerful scripting language, export to XML for external processing and offers the possibility to couple external applications.

External coupling with ADOxx enables the realisation of additional functionality on platform level that is not covered by the core functionality provided by the tool[7] (Fig. 2.1). In order to implement that, we use the AdoScript macro language of ADOxx that is designed for this purpose and allows significant extension possibilities with low programming effort. AdoScript enables integration via a so-called "Message-Port Concept", where specific ports are assigned to each kind of message and the resulting messages are used for further usage and application[8].

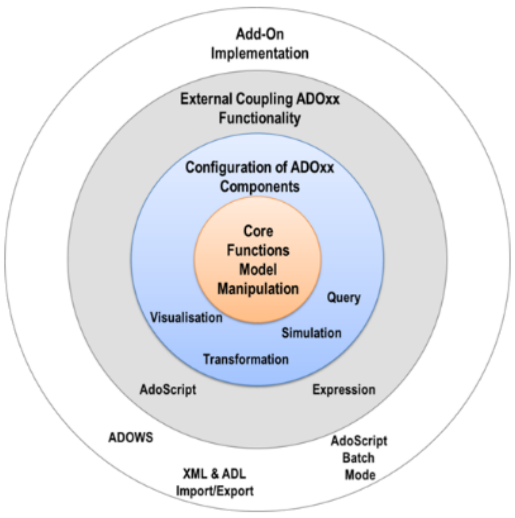


Figure 2.1: ADOxx External Coupling Functionality

AdoScript can be executed on many different ways, so it can be used where it is needed:

- **As menu entry**: used for manual execution (e.g. Scripts 2)
- **In events**: If specific actions are executed, an AdoScript can be automatically called
- **In the Notebook via Programcall**: similar to menu entries, but triggered from within the Notebook.
- **Automatically over Command Prompt**: trigger during startup of ADOxx and handover of AdoScript through the command prompt.
- **From AdoScript Shell**: as a debugging and development facilities to test code snippets (Script 3).

[7]`https://www.adoxx.org/live/external-coupling-overview`
[8]`https://www.adoxx.org/AdoScriptDoc`

Chapter 3

Related Work

In the previous chapter, we described the main components of this thesis; Event Calculus and the ADOxx metamodelling platform. As this thesis is targeted towards the integration of those two components in order to implement a visualisation tool, this chapter studies in more detail other works that we deemed as related to ours. Other visualization tools have been made for many purposes and they have pointed out the advantages of visualisation. Moreover, extensive work with Answer-Set programming has pointed out the capabilities of the language.

3.1 OMiLAB Modelling Method Projects

There exists a big variety of modelling toolkits implemented with the ADOxx Metamodelling Platform. Most of those modelling method projects are available on the OMiLAB website and have been extensively documented [17].

The members of the OMiLAB Network have studied the use of modelling tools in education. Educational activities within OMiLAB address the pragmatics of modelling for all user groups as well as modelling method engineering. The openness of tools and materials enables the worldwide uptake/integration of community results in formal and informal educational activities. The primary target groups are universities, training facilities and similar institutions [12].

The rest of this section, describes some of the most known modelling method projects available within OMiLAB, which are a good reference of work and provide some helpful insight towards the capabilities of the ADOxx platform that we can use for the implementation of our tool.

The Fundamental Conceptual Modelling Languages (FCML) method [15] and its proof-of-concept Bee-Up Tool[1] are aimed at being used as a multi-purpose and multi-layered modelling approach, where method agility is manifested by a multitude of notation alternatives in a single tool for different kinds of users, and also by machine interpretable semantics on which functionality of varying specificity may be built. The method provides the starting steps towards the design

[1]`http://austria.omilab.org/psm/content/bee-up/info`

of domain-specific modelling languages, as well as, a resource of lessons learned which can support both teaching activities in the area of conceptual modelling and scientific experimentation at meta-modelling level.

One version of the entity-relationship languages is the Higher order Entity-Relationship Modelling Language (HERM). Using HERM for database development has important advantages over other extended entity-relationship models yet, it is not suitable for a schema that consists of a large set of entity types with a small set of attributes and a few tuples inside the entity classes. Kramer and Thalheim [20] demonstrate the creation of a graphical modelling tool based on ADOxx which creates a graphical HERM schema by allowing an automatic translation into a logical model after the modelling based on directives which represent a first step on a compiler approach translation.

The Knowledge Work Designer [13], a modelling tool for flexible decision-aware business processes. It is based on two principles: the separation of business logic and process logic and the support of both structures and unstructured knowledge. Process logic can be represented as a structured business process using BPMN, as a non-structured case plan in CMMN or as a combination of both called BPCMN. Decision tables are currently the only representation formalism for structured business logic. Any other business logic can be stored in a file (business data or any kind of document) and referenced via the document model. Future versions of the tool will include support for other types of visual knowledge representation like class diagrams, semantic networks, or ontologies.

3.2 VizDSL: Interactive Information Visualization

Visualization techniques are used as part of Model-Driven Engineering (MDE) to visualise the code, the problem domain and the models used to describe the domain. Morgan et. al. [26] introduced a *platform-independent* and extensible modelling language, VizDSL, which allows non-IT experts to describe, model and create interactive visualizations, quickly and easily. VizDSL is based on the Interaction Flow Modeling Language (IFML) for creating highly interactive visualization. It can be used to model, share and implement interactive visualization based on model-driven engineering principles.

Since VizDSL is platform-independent and extensible through its UML profile, it is important to provide IFML extension details. VizDSL takes a model-based approach rather than a procedural approach to the design process, to meet integration and interoperability requirements in the context of MDE.

The tool will evaluate its usability in terms of *satisfaction, efficiency* and *effectiveness* by means of user studies with users taken from the OGI Pilot[2].

For the implementation of ECAVI, we studied the techniques used by VizDSL and incorporated those that seemed fitting for our purpose.

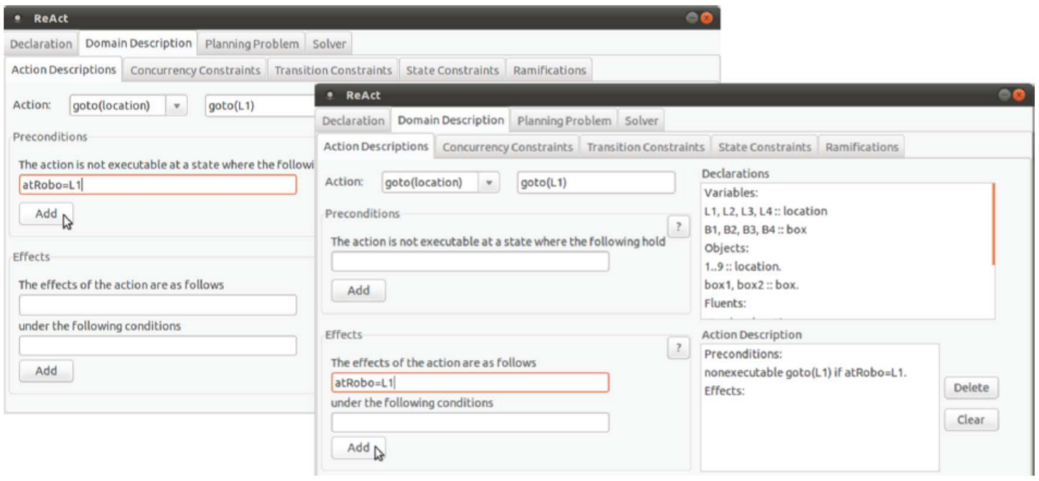
[2]`http://www.mimosa.org/oil-and-gas-interoperability-ogi-pilot`

3.3 ReACT! Interactive Educational Tool

It is of the essence for the robotic systems to be provided with high-level cognitive capabilities since the complexity of the tasks and the variability of environments place high demands on the robots' intelligence and autonomy.

Dogmus et. al. [4] presented an interactive educational tool for artificial intelligence (AI) planning for robotics. ReACT! enables students to describe robots' actions and change in dynamic domains via interactive user interface without first having to know about the syntactic and semantic details of the underlying formalism (Fig. 3.1). They also can solve hybrid planning problems using state-of-the-art reasoners for hands-on applications of cognitive robotics without having to know about their input/output language or usage.

The teaching of AI planning in robotics class for students from various departments and with different backgrounds can be a very challenging and time-consuming work. The job of the tool is to guide the students towards the representation of dynamic domains generically and the solving of planing problems using various planners/reasoners, without having to know the particular specifics.



Action: goto(location) goto(L1)

Preconditions

The action is not executable at a state where the followi

atRobo=L1

Effects

The effects of the action are as follows

under the following conditions

ReAct

Action: goto(location) goto(L1)

Preconditions

The action is not executable at a state where the following hold

Effects

The effects of the action are as follows

atRobo=L1

under the following conditions

Declarations

Variables:
L1, L2, L3, L4 :: location
B1, B2, B3, B4 :: box
Objects:
1..9 :: location.
box1, box2 :: box.
Fluents:

Action Description

Preconditions:
nonexecutable goto(L1) if atRobo=L1.
Effects:

Figure 3.1: Screenshot of the `ReACT!` user interface.

3.4 Sealion IDE for ASP

SeaLion [3] is an Integrated Development Environment for Answer-Set Programming (ASP) [28]. It is developed as part of an ongoing research project on methods and methodologies for developing answer-set programs.

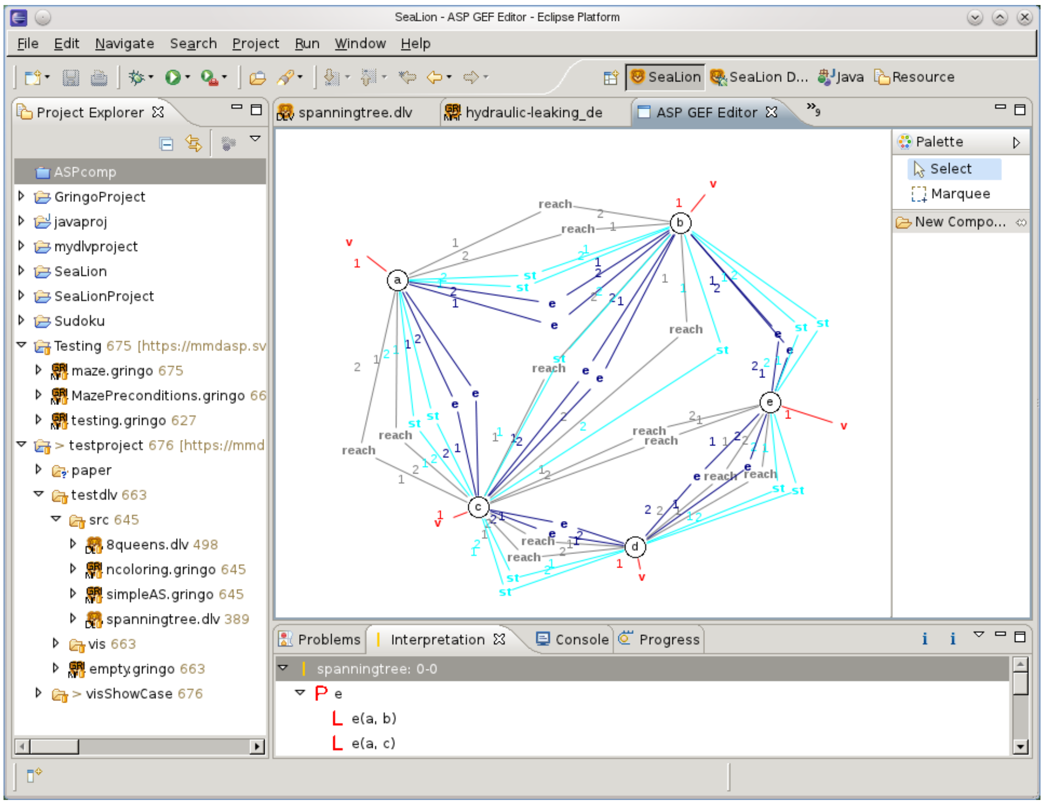
SeaLion is designed as an Eclipse plugin, providing useful and intuitive features for ASP and targets both experts and software developers new to ASP, but with

[3]`http://www.sealion.at/`

familiarity with support tools as used in procedural and object-oriented programming. The goal is to fully support the languages of the state-of-the-art solvers Clasp and DLV, as opposed with other IDEs that support only a single solver.

The IDE is in an alpha version that already implements important core functionality. The editor provides syntax highlighting, syntax checks, error reporting, error highlighting, and automatic generation of a program outline. There is functionality to manage external tools such as answer-set solvers and to define arbitrary pipes between them, as needed when using separate grounders and solvers. Moreover, in order to run an answer-set solver on the created programs, launch configurations can be created in which the user can choose input files, a solver configuration, command line arguments for the solver, as well as output-processing strategies. Answer sets resulting from a launch can either be parsed and stored in a view for interpretations, or the solver output can be displayed unmodified in Eclipse's built-in console view.

The visualisation functionality of SeaLion is itself represented in an Eclipse plugin, called Kara [18]. `Kara` is a tool for the graphical visualisation and editing of interpretations (Fig. 3.2). It is started from the interpretation view. One can select an interpretation for visualisation by right-clicking it in the view and choose between a *generic visualisation* or a *customised visualisation*.



```
spanningtree: 0-0
  P e
    L e(a, b)
    L e(a, c)
```

Figure 3.2: Screenshot of the `SeaLion`'s visual interpretation editor.

Chapter 4

Methodology

In the following chapter we describe the main goals that ECAVI aims to satisfy, with a close relation to the kinds of users we focus on at this point, as well as, the modelling method that we followed and the meta-reasoning and integrity checks that are implemented so far.

4.1 Requirement Analysis

The process, as well as the outcome, of knowledge engineering can benefit by following certain guidelines and good practices, especially in complex cases, such as the domains that action languages are focusing on, which are dynamic and incorporate perplex causal relations. According to Mueller [27], any method for automated commonsense reasoning must incorporate 5 main key aspects:

- The **representation** of commonsense knowledge about the world and real scenarios in it.
- The representation of objects, agents, time-varying properties, events and time, otherwise referred to as **commonsense entities**.
- Dealing with object identity. Representation and reasoning about **commonsense domains**, such as time, space and mental states.
- Address of the *commonsense law of inertia*, release from this law, concurrent events with cumulative and canceling effects, non-deterministic effects, preconditions, and triggered events. Those phenomena are referred to as **commonsense phenomena**.
- Use of representations of scenarios and commonsense knowledge to specify processes for **reasoning**. Specifically, support of default reasoning, temporal projection, abduction, and postdiction.

We incorporate these aspects in our tool with the use of the Event Calculus syntax and semantics and by separating the design into 4 sub-models, further described in Section 4.2.

Our aim is for this tool to be used as an assistant for teaching the fundamental concepts of reasoning about actions and change. For this purpose, at this point of the tool's development, we focus on a target group of novice users and students that are working towards understanding the Event Calculus.

Before we begin to identify the requirements for our tool, we must first distinguish each group of users we focus on and what are their distinct needs. We separate our first target group of users into 4 sub-groups:

1. Users with some knowledge and a little experience in logical programming (e.g. Prolog) but no previous knowledge in Event Calculus.
2. Users with no knowledge in logical programming but with programming background (e.g. knowledge of C, C++, Java etc.)
3. Users with no programming knowledge whatsoever but with some experience in modelling (e.g. people who have worked with other modelling tools implemented on ADOxx such as the Business Process Management tool, ADONIS).
4. Users with no programming knowledge neither any modelling experience.

	User Group 1	**User Group 2**	**User Group 3**	**User Group 4**
Walk-through wizard/tooltips to help make the first steps into the tool making the first model.	X	X		X
Tooltips that explain what each object of the design represents.	X	X	X	X
Help to become acquainted with specific features of the language (eg. delayed effect axioms)		X	X	X
Assistance in the process of building an axiom, avoid syntactical errors.	X	X	X	X
Build ASP programs without needing to know how the answer-set solver works.			X	X

Table 4.1: List of requirements specified for each distinct group of users in our target group

At the above table we pinpoint the needs for our target group of users. Notice that, in a couple of cases a certain target group isn't considered to necessarily have a certain need. For example, a user that has previous experience with other modelling tools made on the ADOxx platform (eg. ADONIS) may not need tooltips that show how to make the first model.

We also want to help the users that already know how Event Calculus works (e.g. knowledge engineers) visualize their programs. Those users, however, will need the support of further features from our tool such as non-determinism, indirect effects of events and others. The need for our tool to support those users as well, is considered part of our future work (see Section 6.2).

In addition, for the knowledge engineer that wants to interoperate with different teams in the same project, quickly communicating the high-level behaviour of a component without delving into the code details (by exchanging visual representation of the model), can be a key aspect in promoting productivity.

4.2 Modelling Method

Karagiannis and Kühn in 2002 [16] proposed a framework for the description of modelling methods (see Fig. 4.1). In this framework a *modelling method* is composed of a *modelling technique* and *mechanisms and algorithms*. The modelling technique is further divided into a *modelling language* and a *modelling procedure* [8, 15].

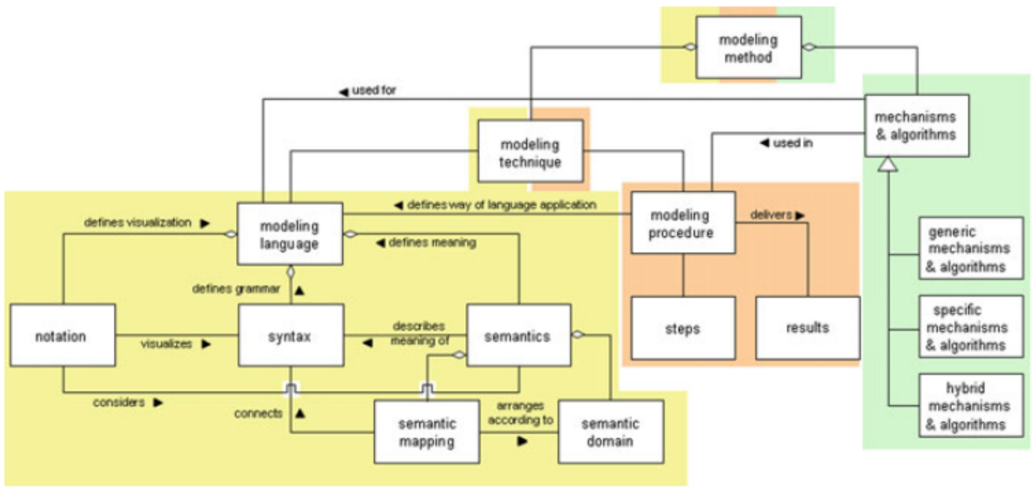


Figure 4.1: The Generic Modelling Method Framework

4.2.1 The ECAVI Modelling Language

In order to build our modelling tool, the modelling language of Event Calculus needs to be realized. As described in Section 2.4, a modelling language consists of:

1. **Syntax:** the specification of a modelling construct
2. **Semantic:** the definition of the meaning of a modelling construct
3. **Notation:** the graphical representation of a modelling construct

A modelling construct can be a (concrete) class, relation class, modeltype or attribute.

For the realization of our modelling language, the steps are clear[1]. First, we define the conceptual aspects of the implementation of our modelling language (namely the Event Calculus in our case), and then we move to the realization of the Event Calculus meta-model with ADOxx.

4.2.2 Conceptualization of the Modelling Language

During this phase, we find a mapping between the generic ADOxx Meta2Model (Fig. 4.2) and our modelling language, the Event Calculus.

[1]`https://www.adoxx.org/live/modelling-language-implementation-on-adoxx`

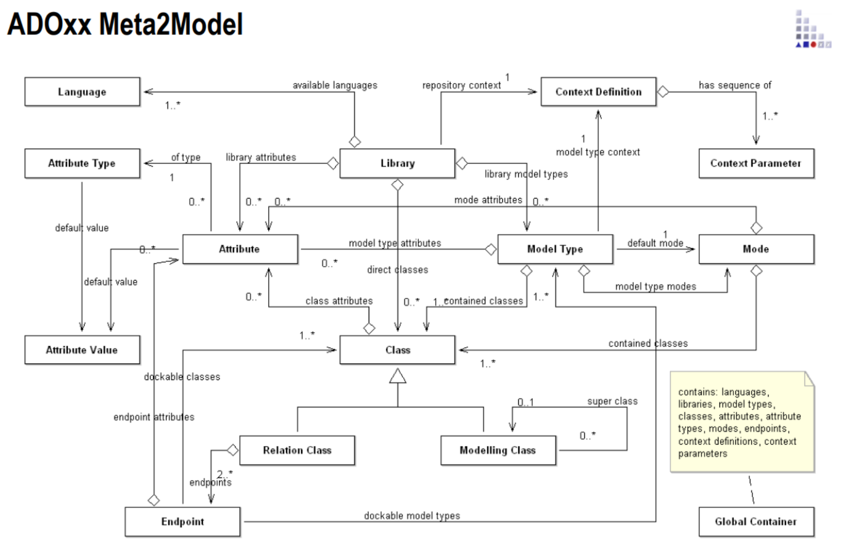


Figure 4.2: The generic ADOxx Meta2Model

In ADOxx there are 3 types of classes[2]:

- Pre-defined Abstract Classes derived from the ADOxx meta model classes and implemented on platform level. These classes have a given semantic and basic syntax in form of attributes.
- Abstract Classes as self-defined classes enabling to structure the meta model and define syntax in form of attributes and semantic, which is inherited by sub-classes. They inherit their behaviour from their super-class - which is often a pre-defined abstract class from the ADOxx meta model.
- (Concrete) Modelling Classes that can be used, when applying the corresponding modelling language. Hence, all model objects created in every model on ADOxx are an instance of a class.

First, we need to consider what classes and relation classes are needed in order to represent the main constructs of a given language definition. Then, we need to think of appropriate super classes (provided by the ADOxx Operationalizable Meta Model) for new classes, as well as, accounting for the definition/configuration of (new) attributes so that our meta model describes the full syntax and semantics of our modelling language. Not all constructs that are part of the syntax of a modelling language need to have a graphical representation; these constructs are usually *abstract* (i.e. not instantiated in models) and are typically for reusability of semantics (e.g. property inheritance). After that, we are ready to define an intuitive graphical notation for the classes and relation classes in order to simplify the modelling, with the use of the GraphRep class attribute (see Section 2.4.1).

[2]`https://www.adoxx.org/live/classes`

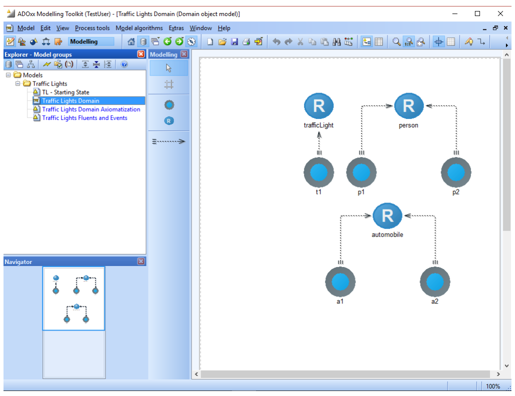
For our implementation, we make use of the pre-defined classes of the Static Library in order to define our Domain Objects, the constants of our domain. The Dynamic Library is then used to define models that describe the rest of our domain.

Relations between two objects of the same model are defined with a corresponding instance of the relation class. In order to define relations between objects that belong in different models, a special configuration of a Relation Class is needed, called InterRef. InterRef makes the connections needed for our models to interlink[3].

All the defined classes and relation classes of our implementation are shown in Appendix A.

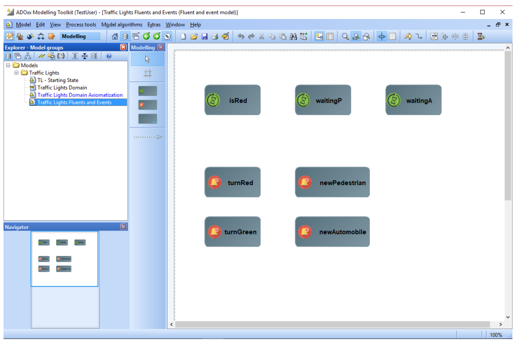
4.2.3 Implementation of the Modelling Language

In ECAVI, we rely on the ASP syntax and semantics (described in Section 2.2), which implement Event Calculus theories. As for the visual notation deployed to model the key aspects of the formal languages, we developed a set of visual cues shown in Figure 4.4



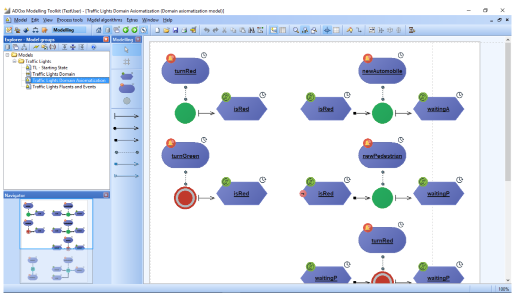
ADOxx Modelling Toolkit (TestUser) - [Traffic Lights Domain (Domain object model)]
trafficLight
person
t1
p1
p2
automobile
a1
a2

(a) Domain Object Model

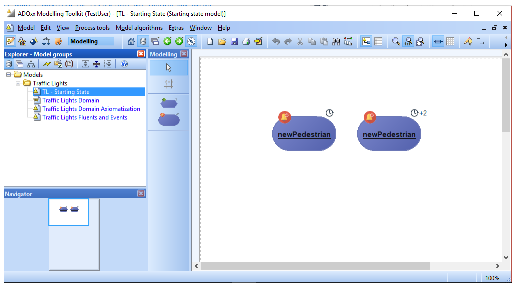


ADOxx Modelling Toolkit (TestUser) - [Traffic Lights Fluents and Events (Fluent and event model)]
isRed
waitingP
waitingA
turnRed
newPedestrian
turnGreen
newAutomobile

(b) Fluent and Event Model



(c) Domain Axiomatization Model



ADOxx Modelling Toolkit (TestUser) - [TL - Starting State (Starting state model)]
newPedestrian
newPedestrian

(d) Starting State Model

Figure 4.3: Screenshots of the 4 model types defined in the ECAVI tool

[3]https://www.adoxx.org/live/relations

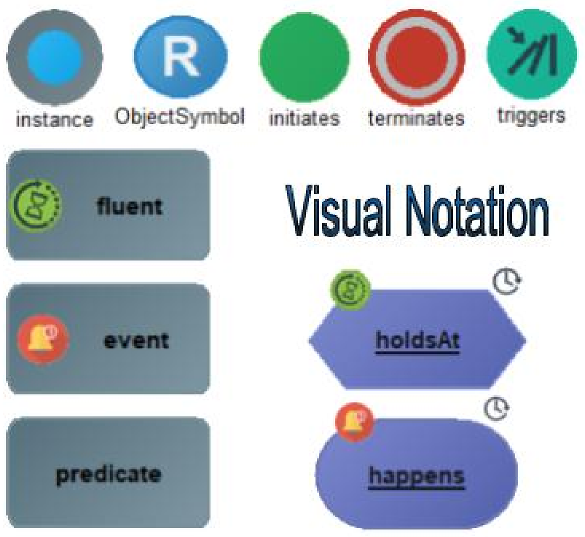


Figure 4.4: Graphical Notation of ECAVI

4.2.3.1 Modelling Procedure

To accommodate the modelling process, we follow a common practice in knowledge engineering for dynamic domains, which breaks down the modelling tasks into four sub-models (Fig. 4.3):

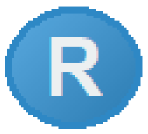
- The *Domain Object* model (Figure 4.3a), which specifies all *Object Symbols* (or Roles) and the *Instances* (or Constants) that populate our domain.
- The *Fluent and Event* model (Figure 4.3b), which specifies all dynamic aspects of our domain, in the form of *fluents*, *events* , and other user-defined *predicates*. Together with the Domain Object model, this part defines the *signature* (or *alphabet*) of our domain axiomatisation.
- The *Domain Axiomatisation* model (Figure 4.3c), which axiomatises the dynamics of our domain. This is the main part of the modelling process, supporting the user in defining effect axioms (*Initiates, Terminates, Triggers*), coupled with preconditions and effects defined in the previous models, e.g., fluent and event expressions.
- Finally, the *Starting State* model (Figure 4.3d) defines the initial state of a domain, and the narrative of events that happen at various timepoints. This is used as input to the **clingo** solver, to find answer sets satisfying the domain dynamics.

With the help of Event Calculus, we can represent commonsense knowledge and scenarios, and use the knowledge to reason about the scenarios (Erik T. Mueller [27]). A full description of the basic notions of the Event Calculus and ASP, that we mapped to graphical representations follows.

4.2.3.2 Visual Notation

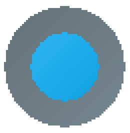
We design the graphical representation of each object with the use of the GraphRep class attribute. The following classes are part of the Static Library:

Object Symbol



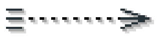
A constant or variable that defines the context/domain of the implementation. An object symbol has an arity defined by the number of its instances.
The symbol `R` represents the fact that an Object Symbol is perceived like a **Role** of an Instance.

Instance



An instance of an object symbol (constant or variable, called term in ASP). An instance is associated with one or more object symbol(s) via a *ISA Association*.

Has Role (Relation)



The connector that defines a relation between an Instance and an Object Symbol. This relation is practically used to give a role to an instance.

Figure 4.5 shows an example of how the Domain Objects are drawn in our model. In this example, *anakin* has the role of **father** and *luke* and *leia* have the role of **child**. All three of them also have the role of a **person**.

The corresponding EC definition is:

```
person(anakin;luke;leia).
father(anakin).
child(luke;leia).
```

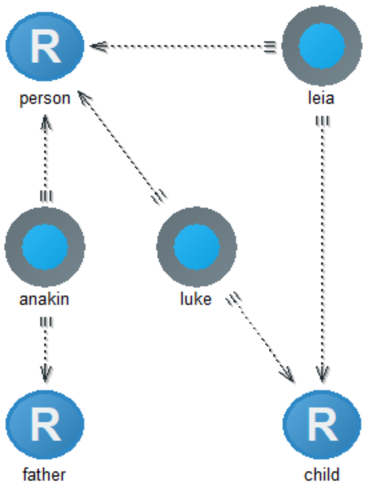


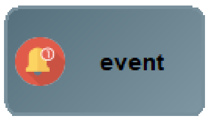
Figure 4.5: Domain Objects Representation Example

The following classes are part of the Dynamic Library:

Fluent

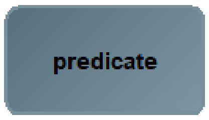
A time-varying property of the world. A fluent has a truth value at a timepoint or over a timepoint interval; the possible truth values are true and false.

Event



An event or action that may occur in the world. It may occur or happen at a timepoint. After an event occurs, the truth values of the fluents may change (i.e. an event may initiate or terminate a fluent).

User-Defined Predicate



Applies a nature to an already defined instance. This type of predicate differs from a fluent in a sense that it is not dependent on time.
Eg. `movableObj(Obj) :- object(Obj).`

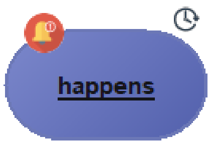
Predicate is a precondition to a fluent or an event (Relation)

This connector defines a relation between a User-Defined Predicate and a Fluent or an Event.
Eg. `fluent(rightOf(X,Y)):-`
`object(X), object(Y) movableObj(X), movableObj(Y).`

HoldsAt

`HoldsAt(f,t)` represents that fluent f is true at timepoint t. A fluent is linked to an instance of the HoldsAt class with the InterRef relation.

Happens



`Happens(e,t)` represents that event e occurs at timepoint t. An event is linked to an instance of the Happens class with the InterRef relation.

Happens and *HoldsAt* can be negated. In ECAVI, we visualize the negation of an Event Calculus predicate with the symbol ¬ with a red background on the left side of the predicate (Fig. 4.7).

For a fluent we use the symbol of the hourglass to visualize the fact that the fluent's state may change from time to time. For an event we use the symbol of a bell notification to imply that an action occurs at the time the event happens. The same symbols are used on the `HoldsAt` and the `Happens` classes, respectively. Both the `HoldsAt` and the `Happens` predicates occur at a certain timepoint and for that purpose we added the symbol of a clock at the top right of their graphics. Notice that, at the cases when an event occurs at some timepoints before or after another (e.g. T-1, T+2, etc.), the corresponding timestamps are shown (Fig. 4.7)

Effect Predicates

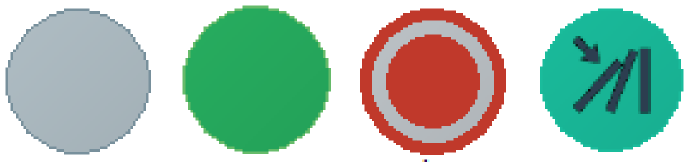


Figure 4.6: States of an effect:
(a) Default (b) Initiates (c) Terminates (d) Triggers

When an effect is created, it assumes the default state (temporary state upon creation). The user then, has to define the type of the effect. The effects of event can be of 3 different types: Initiates, Terminates or Triggers (further described in Section 2.1).

The effects of events may have preconditions that define if the event will have its intended effect (qualification):

- A **fluent precondition** is a requirement that must be satisfied for an event to have an effect. We express the fluent preconditions in the form of HoldsAt.
- An **action precondition** is a requirement that must be satisfied for the occurrence of an event. We express the fluent preconditions in the form of Happens.

Fluent Precondition (Relation)

The connector used for defining fluent preconditions. The relation is outgoing from an instance of a HoldsAt class and incoming to an instance of an Effect.

e.g. Fig. 4.7 shows an example of 2 fluent preconditions to an effect

Event Precondition (Relation)

Similar to the connector for a fluent precondition with the only difference of a bullet at the starting end (in order to distinguish between them with a glance).

e.g. Fig. 4.8 shows an example of an event precondition to an effect

Triggering Event (Relation)

The connector used for linking an effect to the event that has the behaviour of a trigger.
Figures 4.7 & 4.8 display examples of triggering events to effects

Triggered Fluent (Relation)

An effect of type Initiates or Terminates has an outgoing relation to the fluent that will be triggered if all the preconditions are true. This connector specifies this type of relation.
In Fig. 4.8, an example of a triggered fluent is shown on the right side of the effect.

Triggered Event (Relation)

Similar to the previous connector, this connector specifies the outgoing relation from an effect of type Triggers to the event that will be triggered if all the preconditions are true.
In Fig. 4.7, an example of a triggered event is shown on the right side of the effect.

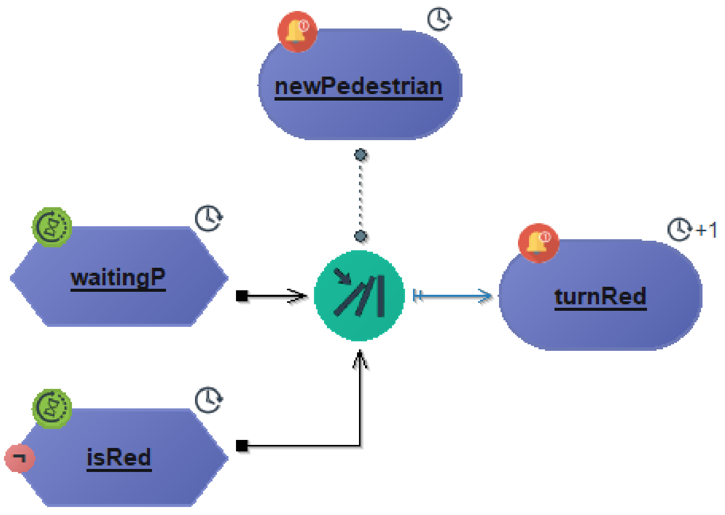


Figure 4.7: Triggering axiom with two fluent preconditions.

Figure 4.7 illustrates an example of a triggering axiom where the effect has the type "Triggers" and in order for the event `turnRed` to be triggered on timepoint T+1, the event `newPedestrian` has to occur at the timepoint T and 2 fluent preconditions (`waitingP` & ¬`isRed`) have to be true as well.

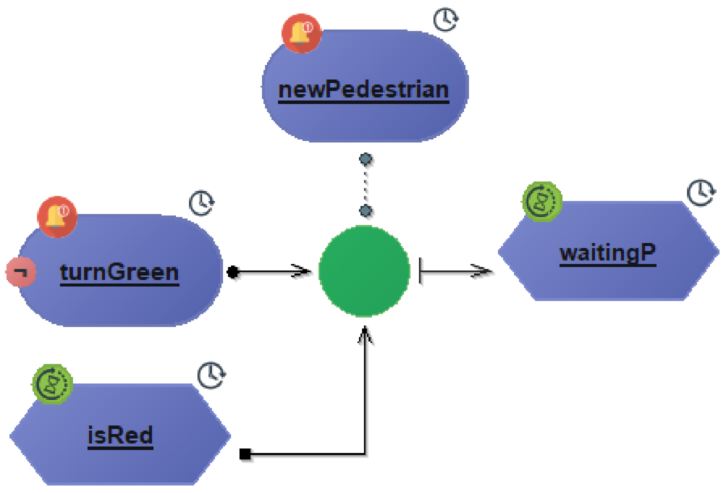


Figure 4.8: Initiates axiom with a fluent and an event precondition.

Figure 4.8 displays an example of an axiom where the effect has the type "Initiates" and in order for the fluent `waitingP` to become true at timepoint T, the event `newPedestrian` has to occur at timepoint T and a fluent precondition (`isRed`) and an event precondition (¬`turnGreen`) have to be true.

For better visualization, the preconditions are shown before the effect, the triggering event is shown above the effect and the triggered event/fluent on the effect's right side. This order and the connectors (arrows) are drawn this way to better represent the flow of time.

Full examples with the implementation of a use case are shown in Section 5.2

4.3 Translation into ASP

The axiom shown in Figure 4.7 when translated into ASP composes the following code:

```
happens (turnRed(t1),T+1) :-
     not holdsAt(isRed(t1),T),
     holdsAt(waitingP(PERSON2),T),
     happens(newPedestrian(PERSON1),T),
     PERSON1!=PERSON2.
```

Meaning:
At a certain timepoint T, if a new pedestrian (PERSON1) arrives at the traffic light (t1) for the purpose of crossing to the other side of the road, and the light is not red and there is already another pedestrian waiting (PERSON2), then the traffic light will turn red at the next timepoint T+1

For simplicity, we assume that there is only one traffic light in this case with two states: when the light is green the cars can move through and the pedestrians have to wait, and when the light is red the cars will have to stop and the pedestrians can cross the road.

In ASP, it is a common practice for predicates and constants to start with a lower-case letter and for variables to start with an upper-case letter (more details in Section 2.2). In our example, *t1* is a constant that has already been defined with the role of a traffic light. *PERSON1* and *PERSON2* are variables of the type person that will each be mapped to an already defined constant of a person when the resulting answer sets are produced by the reasoner. Moreover, T is a variable that represents a timepoint which can equal to an integer. In this example there are **2 fluents**: *isRed* and *waitingP*, and **2 events**: *turnRed* and *newPedestrian.*

In order to achieve the correct translation of the models, we export them into XML format (a feature supported by ADOxx) and then an intermediate Java program performs the required analysis on the XML exported file where for each complex object it finds all the relations of this object (incoming and outgoing), translates them according to the ASP syntax and writes the corresponding ASP rules and axioms into the file that compiles the final ASP program that the Clingo reasoner is gonna run.

As already mentioned in Section 4.2.2, the InterRef functionality enables us to make mappings between objects that belong in different models. For example, each instance of the HoldsAt class (in Domain Axiomatization model) is linked with a fluent from the Fluents and Events Model. These kind of links are essential for the translation into ASP. If any essential InterRef is missing, then our program cannot move on to the translation of the XML extract into ASP code.

To make the process of the translation easier, we implemented Java classes that imprint the full structure of each type of "object" in ASP. An overview of the classes that comprise the XML translator to ASP is shown in Section 5.

Essentially, the Java code reads the XML files in the order that ASP code is commonly written (domain objects, rules and axioms before the definition of the starting state). So, it first reads the Domain Objects Model export, parses the constants and writes them into an ASP file (.lp extension). After that, it reads the Fluents and Events Model export, parses and writes them into the file and then moves on to the Domain Axiomatization Model export and at the end the Starting State Model export. The main class of our program is the class `XMLtoASP`. For each XML file exported from our model it makes the procedures shown in Algorithm 1.

Notable is the way we define which are the variables of the ASP program. Lines 8-12 of the algorithm describe our solution. When an argument has an InterRef to a class type Instant, then it is a constant that has already been defined in the Domain Objects Model, and is written into the ASP program with lower-case letters. Otherwise, if the InterRef points to a Object Symbol class object, then we assume that the argument is a variable of this role and is then written into the ASP program file with upper-case letters.

Algorithm 1 Steps for translating XML into ASP

```
 1: if Domain Object Model then
 2:    for all objects of type Instance do
 3:       get the Instance's role(s)
 4:       WRITE the domain object into the .lp file
 5: else if Fluent and Event Model then
 6:    # fluents and events have the same structure
 7:    for all events and fluents do
 8:       find all arguments by reading the InterRefs to Instances & Object Symbols
 9:       if the InterRef is an Instance then
10:          the argument is an already defined constant
11:       else if it is an Object Symbol then
12:          we assume that the argument is a variable
13:       WRITE fluents/event into the .lp file
14: else if Domain Axiomatization Model then
15:    for all HoldsAt objects do
16:       follow similar steps with before to find the InterRef to the fluent and
          arguments
17:    for all Happens objects do
18:       follow similar steps with before to find the InterRef to the event and
          arguments
19:    for all Effects do
20:       get the relation to the triggering event
21:       if effect type = Initiates or Terminates then
22:          get the relation to the triggered fluent
23:       else if effect type = Triggers then
24:          get the relation to the triggered event
25:       if effect has preconditions then
26:          for all effect preconditions (event and fluent) do
27:             get event and fluent preconditions
28:       WRITE the axiom into the .lp file
29: else if Starting State Model then
30:    for all HoldsAt and Happens that comprise the starting state do
31:       WRITE it into the .lp file
```

4.4 Meta-reasoning & Integrity Checks

The basic idea of ASP is to find solutions to a problem, in the form of answer sets (usually stable models) of a logic program, which consists of rules and constraints that define properties of the solutions. The problem is solved by computing stable models using answer set solvers like clasp [10]. Simple reasoning over answer sets is frequently supported by ASP systems but more specialised reasoning tasks require more processing and are not easily done. In the previous years, there have been some works focused on the job of meta-reasoning on answer sets [6, 29].

The adoption of the logic programming paradigm offers certain leverage to the knowledge engineer, such as the ability to prove properties or to easily find optimal solutions, yet the process of detecting and ironing out logical errors is often cumbersome. This is due to the declarative nature of program execution, which does not follow a procedural execution, but instead relies on logical dependencies among rules in the encoding.

We aim to implement some extensive meta-reasoning into our tool that will help the more experienced users run and visualize their programs better, but at this point of ECAVI's development, we focus on implementing simple meta-reasoning tasks and integrity constraints that are designed for the purpose of helping the user create full and syntactically correct programs.

AdoScript, the macro language of ADOxx, is designed for the purpose of providing the meta modeler with significant extension possibilities with low programming effort (see Section 2.4.2). We make use of AdoScript for implementing a number of features that make the user's work with building axioms easier and also enable us to support a number of integrity checks.

In ECAVI, we implement simple but fundamental checks for syntactical errors either with the help of AdoScript or in Java at design time before the run of the Clingo reasoner. In more detail:

- **An instance must always have at least one role** (object symbol). For this purpose, the Java programs checks the Domain Object Object for whether an instance isn't mapped to any Object Symbol.
- **Whatever the type of an Effect (Initiates, Terminates or Triggers) there must always be an event that triggers this effect.** For this purpose, we developed a script in AdoScript that automatically generates the triggering event for an Effect (see Script 4).
- **An effect, with the type of Initiates or Terminates requires a fluent to be triggered.** So it must always have an outgoing relation to a fluent. **And if an effect is of type Triggers it must always have an outgoing relation to an event to be triggered.** Similarly with the triggering event, a script is developed for the auto-generation of the corresponding objects.
- **In order for an ASP program to run on the Clingo reasoner, the starting state of the world of the designed domain has to be defined.**

> At the start of the script that implements all the main functionality of the ADOxx External Coupling with Java and Clingo (Script preview 6) we check whether the script was called from a Starting State Model. This model has to be designed before we can move on to the translation into ASP.

Preconditions are not necessary for an effect. However, effects often have one or more preconditions. A script was composed for the purpose of making the process of generating a new precondition for an effect easier and quicker (Script 5). The user has to right click on the effect the desired precondition is to be linked and then choose from the context menu what kind of precondition it is gonna be (fluent or event precondition).

The objects that are automatically generated are empty instances. The user then has to open the object's notebook and define the corresponding relations (i.e. InterRef) and information. With this process, we point the user to the right way of modelling an axiom, minimizing as well this type of syntactical errors.

Furthermore, the ADOxx meta-modelling platform has some build in functionalities that enable us to enforce some conditions that are essential for building rules in EC, like unique name assumptions and having the preconditions and all events and fluents whatsoever already defined when applying them on an axiom.

After the user clicks the "Run" option and chooses which models define his/her program, those models are then exported into XML and the Java intermediate program checks if there is an object with a missing relation before doing any other activity (eg., an Instance has no Role, a HoldsAt precondition has no InterRef to a fluent etc.). If any missing relation is found, then an error is raised pointing the user to the object that is undefined, highlighting it as well.

A sample list of scripts is shown in Appendix B.

Chapter 5

Implementation & Use Case

In this chapter we describe how our tool was implemented. We present the architecture of the developed system specifying the role that each component has and how everything comes together. We also present a basic use case scenario that was designed as a sub-problem to a much more extensive real-life problem.

5.1 Architecture

ECAVI is developed with the use of the ADOxx metamodelling platform. We try to develop a modelling language that is tailored to the Event Calculus way of representing causal relations, making use of the ADOxx External Coupling functionality with the AdoScript macro language (see Section 2). A high-level overview of the tool's architecture is shown in Figure 5.1.

The ADOxx meta-modelling platform comprises of the *Development Toolkit* and the *Modelling Toolkit*. The whole realization of the ECAVI meta-model happens on the Development Toolkit, which we use in order to build the modelling language of our tool, by defining our modelling constructs (classes, modeltypes and attributes) stepping on some ADOxx pre-defined abstract classes and to define our modelling procedure by separating the process of building a full ASP program into 4 sub-models (see also Section 4.2). We design the graphical representation for each class and relation class by defining the class attribute GraphRep and with the help of the GraphRep online repository (see Section 2.4.1). Furthermore, we use the Development Toolkit to realise the external coupling that provides the tool with additional functionality, with the help of the AdoScript macro language (see Section 2.4.2).

The AdoScript Message-Ports and Commands (see Appendix C) enable us to define new menu entries, realize specific model checking and provide additional add-on-programming. The scripts 2 and 3 in Appendix B give an example of how AdoScript provides add-on functionality. AdoScript is the actual link between ADOxx and Clingo following these steps:

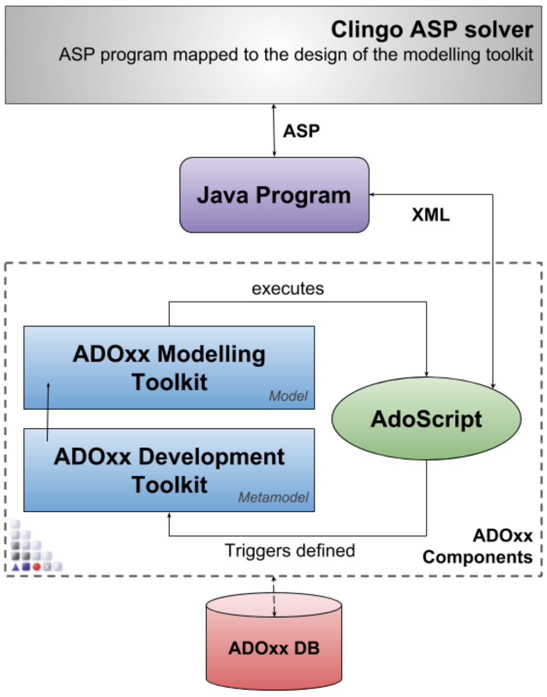
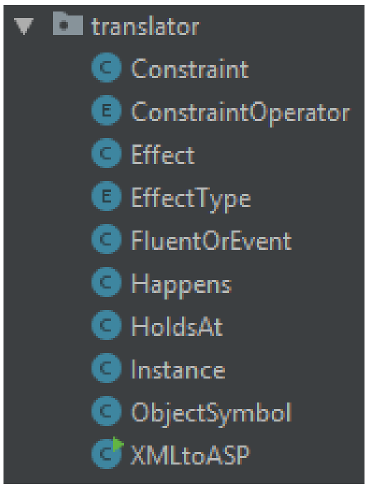


Figure 5.1: The architecture of the ECAVI modelling toolkit

1. Export the designed models in XML.
2. Provide the XML files as input to the Java program that implements the translation of the designs into ASP programs.
3. Run the translated ASP program on the Clingo reasoner and save the resulting answer sets into a file.
4. Provide the Clingo results as input to the Java program that parses them into XML format which is needed for the results to be displayed back into the model.

In the Modelling Toolkit, the end user makes use of the modelling language that was realised in the Development Toolkit. The user will follow the modelling procedure steps in order to design a new domain of application that will be translated into an ASP program and given into the Clingo reasoner to produce the resulting answer sets. During the user's work in the Modelling Toolkit, the various functionalities implemented in AdoScript and defined in the Development Toolkit will be triggered either consciously by the user (e.g. when a user chooses to add a precondition to an Effect from its context menu) or automatically when another event happens and causes the functionality to be triggered (e.g. when an Effect is created and its type is chosen, a couple of object instances are automatically created and linked with it).

The Java intermediate program is made for the purpose of translating the designed models into an ASP program and vice versa. On the way to making the translation from XML to ASP and vice versa, easier and more efficient, we implemented Java classes that imprint the full structure of each type of "object" in ASP. In more detail, each model designed in ADOxx is exported as an XML file. The Java program parses each one of the XML files and with a certain order that follows the common practice of writing ASP programs (first constants, then events and fluents, then the axioms and at the end the starting state), translates them into ASP and writes them into the file that the Clingo reasoner runs and produces the desired answer sets. It then parses the results and produces the XML file that is read by AdoScript and displayed back into the model. The algorithm that the Java uses in order to translate the XML into ASP is shown in Section 4.3.



```
translator
    Constraint
    ConstraintOperator
    Effect
    EffectType
    FluentOrEvent
    Happens
    HoldsAt
    Instance
    ObjectSymbol
    XMLtoASP
```

Figure 5.2: Overview of the classes of the Java translator program.

Figure 5.2 shows an overview of the Java classes that were implemented. Each ADOxx object class is mapped with a corresponding class in Java. In addition, we added 2 enumerations for the types of an effect (Fig. 5.3) and for the constraint operator types (Fig. 5.4). The constraints are defined for predicate arguments, for example in section 4.3, the 2 instances of type person are part of the constraint `PERSON1 != PERSON2`.

```java
public enum EffectType { DEFAULT, INITIATES, TERMINATES, TRIGGERS }
```

Figure 5.3: Effect types enumeration

```java
public enum ConstraintOperator { NON_EQUAL, EQUAL, GREATER, GREATER_OR_EQUAL, LESSER, LESSER_OR_EQUAL }
```

Figure 5.4: Constraint types enumeration

The following script is an excerpt of the AdoScript code that has the functionality of adding a new menu item under a new top-level menu called "Menu algorithms". This menu item called "Run..." calls the script where the whole process of the communication between ADOxx and clingo is implemented with the help of the Java program. An excerpt of the script that implements the whole run process is shown in Appendix B.

Algorithm 2 Add new menu item for the Run process

```
# add the item as a new menu item under a new top-level menu called
    "Model algorithms" for each component
ITEM "Run..."
    acquisition:"Model algorithms" modeling:    "Model algorithms"
    analysis:   "Model algorithms" simulation:  "Model algorithms"
    evaluation: "Model algorithms" importexport:"Model algorithms"

# execute an external ASC file when clicking on the menu, could be
    in a file space (as below) or also in library using db:\\
EXECUTE file: ("D:\\Nena\\AdoScript\\runClingo.asc")
```

The Script 3 shows the process of adding a new menu item, under the top-level menu "Extras", that opens a Debug Shell where the user can enter AdoScript code that he wants to test out. This functionality was mostly designed for helping us test out the code we added for the functionalities we implemented, but can be helpful, as well, for users that already have gained experience with the tool and want to use AdoScript to "play" with their models.

Algorithm 3 Add AdoScript Debug Shell

```
ITEM "AdoScript Debug Shell"
    acquisition: "Extras" modeling:     "Extras"
    analysis:    "Extras" simulation:   "Extras"
    evaluation:  "Extras" importexport: "Extras"
IF (type (adoscript) = "undefined"){
     SETG adoscript:""
}
CC "AdoScript" EDITBOX text:(adoscript)
     fontname:"Courier New" fontheight:12
     title:   "Enter the code you want to test..." oktext:"Run"
IF (endbutton = "ok"){
     SETG adoscript:(text)
     EXECUTE (text)
}
```

5.2 An Example of Application - Use Case

In this section, an example of application (ie. a *use case*) is described. Even though we made the use case intentionally trivial, it can be generalized to account for more complex domains with larger knowledge bases.

As part of a general-purpose smart city project, an engineer wishes to model the behaviour of a particular type of traffic lights that change from red to green and back according to some rules. The desirable behaviour is for the light to stay green for cars as long as a predefined number of pedestrians show up and wait to cross the road. The idea is to model the dynamics of the traffic light domain with a given ruleset, so that it can be integrated in the overall smart city system and be stress-tested through simulation to fine-tune its parameters.

Figures 5.5, 5.6, 5.7, 5.8, 5.9, 5.10, 5.11 describe the creation of our use case step-by-step. Beside each figure, we also show the corresponding ASP code.

```
trafficLight(t1).
person(p1).
person(p2).
automobile(a1).
automobile(a2).
```

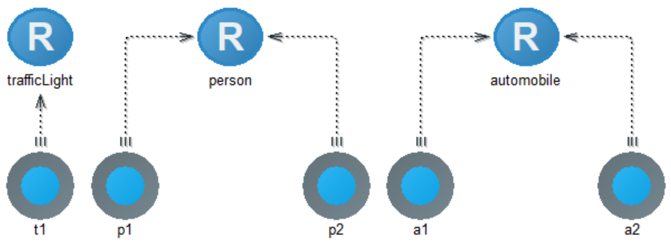


Figure 5.5: Domain Objects of Traffic Light example

Figure 5.5 shows how the Domain Objects of our problem are designed. We have defined an instance of a traffic light (*t1*), two instances of type person (*p1*,*p2*) and two more instances of type automobile aka. cars (*a1*,*a2*).

```
fluent(isRed(TRAFFICLIGHT)) :-
     trafficLight(TRAFFICLIGHT).
fluent(waitingP(PERSON)) :-
     person(PERSON).
fluent(waitingA(AUTOMOBILE)) :-
     automobile(AUTOMOBILE).
event(turnRed(TRAFFICLIGHT)) :-
     trafficLight(TRAFFICLIGHT).
event(turnGreen(TRAFFICLIGHT)) :-
     trafficLight(TRAFFICLIGHT).
event(newPedestrian(PERSON)) :-
     person(PERSON).
event(newAutomobile(AUTOMOBILE)):-
     automobile(AUTOMOBILE).
```

Figure 5.6: Fluents and Events of the Traffic Light example

The definition of the fluents and the events of the traffic light use case is demonstrated in Figure 5.6. For our problem we have 3 fluents which define the state of an instance (the top 3 objects of the figure with the symbol of the hourglass). The first fluent's argument is an instance of the *trafficLight* role and is used to *describe if the traffic light is red.* The second fluent, has a person as its argument and is used to define if the person is *waiting to cross the street.* And, finally, the third fluent describes the state where a *car is waiting for the light to turn green again.*

Moreover, we define 4 events which describe actions that occur at a certain time (the bottom 4 objects in the figure with the symbol of the bell). The first 2 events describe the action of a *traffic light becoming red* and a *traffic light becoming green.* The third event happens when *a new person arrives* at the stop for the purpose of crossing the road and the last event describes the action of *a new car arriving* at the location the traffic light is.

Fluents and events may contain both variables and instances as arguments. They may also contain other predicates, where in this case, a user-defined predicate object is made and connected with the fluent/event using the relation/connector shown in Section 4.2.3.2.

```
initiates(turnRed(TRAFFICLIGHT),
          isRed(TRAFFICLIGHT), T):-
     trafficLight(TRAFFICLIGHT), time(T).


terminates(turnGreen(TRAFFICLIGHT),
          isRed(TRAFFICLIGHT), T):-
     trafficLight(TRAFFICLIGHT), time(T).
```

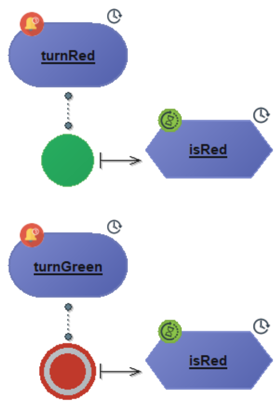


Figure 5.7: Simple effects of Traffic Light example

In Figure 5.7 the "simple" effects with no preconditions of our problem are defined. At the top of the picture, an initiates effect is displayed and at the bottom, a terminates effect is displayed that affect the state of a traffic light. The meaning of those effects is: at a certain time T if the event *turnRed* occurs then the traffic light will turn red and respectively, at a time T if the event *turnGreen* occurs then the traffic light will turn green.

Figure 5.8 demonstrates 3 effects that have preconditions. The first one, at the top of the picture, is a positive effect axiom where, at a certain time, if the traffic light is red and a new car arrives, then the car has to wait (for the light to turn green). The second positive effect axiom describes the behaviour where, if the traffic light is not red (ie. if it's green and the cars can move through) at a certain time and a new pedestrian arrives, then the pedestrian has to wait. Notice that in this axiom, the `holdsAt` precondition which is negated in the ASP code, also has the negation sign visualized with a red background on the left side of the holdsAt object.

```
initiates(newAutomobile(AUTOMOBILE),
          waitingA(AUTOMOBILE), T) :-
     holdsAt(isRed(TRAFFICLIGHT),T),
     automobile(AUTOMOBILE).

initiates(newPedestrian(PERSON),
          waitingP(PERSON),T) :-
     not holdsAt(isRed(TRAFFICLIGHT),T),
     person(PERSON),
     trafficLight(TRAFFICLIGHT),
     time(T).

terminates(turnRed(TRAFFICLIGHT),
          waitingP(PERSON),T) :-
     holdsAt(waitingP(PERSON),T),
     trafficLight(TRAFFICLIGHT),
     person(PERSON),
     time(T).
```

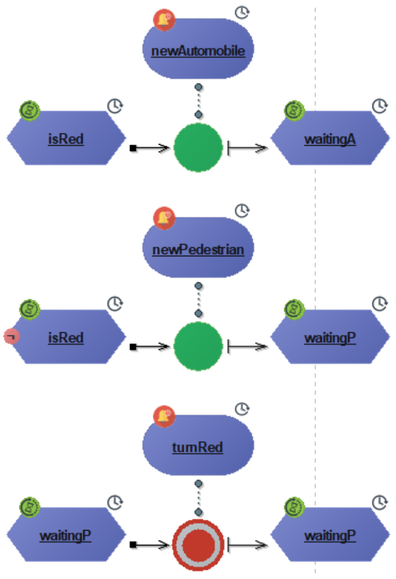


Figure 5.8: Effects with preconditions from the Traffic Light example

At the bottom of the picture, a negative effect axiom is shown, a terminates effect which shows that, at a certain time, if a person is waiting to cross the street and the traffic light turns to red, then the pedestrian will no longer wait and can cross the street.

```
happens(turnGreen(TRAFFICLIGHT),T+1):-
    happens(turnRed(TRAFFICLIGHT),T).
```

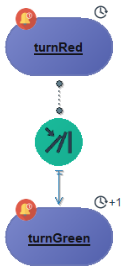


Figure 5.9: Simple trigger axiom

In the Figure 5.9 we can see a simple trigger axiom which describes the transition of a traffic light's state from one timepoint to another. In particular, if the traffic light was red at time T, then on time $T+1$ the traffic light will become green.

```
happens (turnRed(t1),T+1) :-
    not holdsAt(isRed(t1),T),
    holdsAt(waitingP(PERSON2),T),
    happens(newPedestrian(PERSON1),T),
    PERSON1!=PERSON2.
```

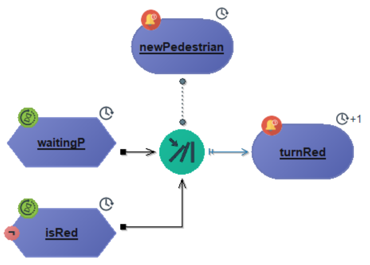


Figure 5.10: Trigger axiom with preconditions.

The Figure 5.10 demonstrates the same axiom that we analysed in Sections 4.2.3.2 and 4.3. If describes the effect: if a new pedestrian arrives at the traffic light, at time T, and the light is not red (ie. it is green for the cars) at the same time, and another person is already waiting to cross the street, then the traffic light will turn red at the time T+1.

```
happens(newPedestrian(p1),0).
happens(newPedestrian(p2),2).
```

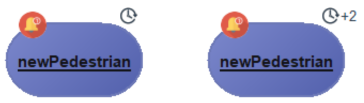
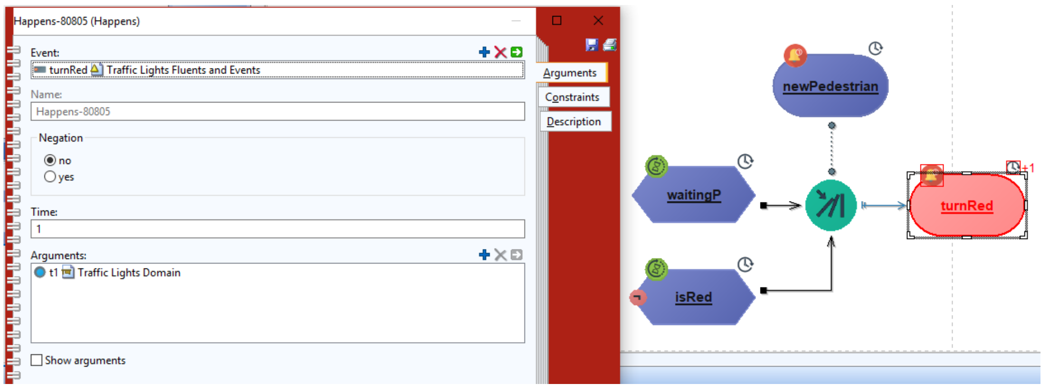


Figure 5.11: Starting state objects.

In Figure 5.11 we can see the 2 `happens` objects that the starting state of our problem consists of. The starting state is initialized with the appearance of the person `p1` at time 0 (`T`) and then the appearance of a second person `p2` at time 2 (`T+2`).

The graphic representation of the axioms aims to visualise a certain flow of actions. In particular, the effects have incoming arrows from the fluents and the events that are considered as preconditions to them. The connector between the events that are the triggering point for an effect to take place (if all the preconditions are met) and the corresponding effect, is drawn with a bullet on each edge, and the fluent or event that is the result of the effect has an arrow pointing to it.

The ADOxx Notebook is where the user can interact with an instance of a class and is accessed with a double-click on the object. In the object's notebook, the user can give values to the attributes of each class. For each different type of class, we define which attributes are part of the notebook (e.g. all the InterRefs are defined in the notebook). The Notebook can be divided into chapters and the attributes can be grouped inside chapters.


Happens-80805 (Happens)

Event:
turnRed Traffic Lights Fluents and Events

Name:
Happens-80805

Negation
[✓ no] yes

Time:
1

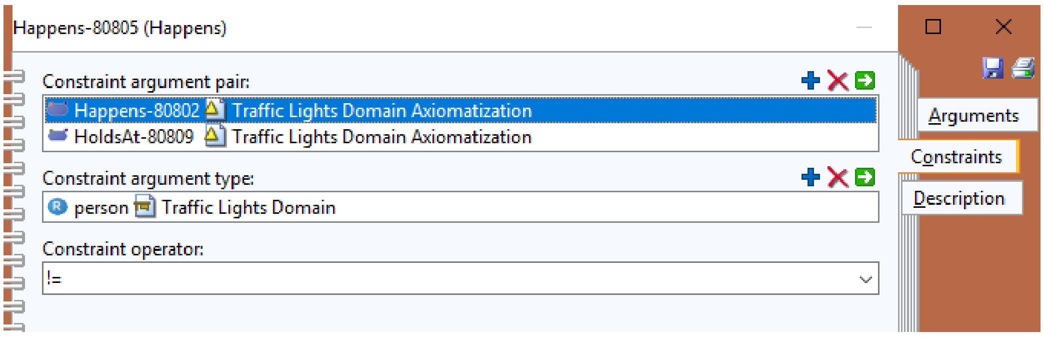
Arguments:
t1 Traffic Lights Domain

Show arguments

Figure 5.12: Example of a notebook for the class Happens

An example of the notebook for the class `Happens` is demonstrated. This object is the triggered event of the Trigger axiom shown in Figure 5.10. The Notebook of the class Happens is divided into 3 chapters. Figure 5.12 displays the first chapter, where the user is called to define the InterRef to the event to be mapped to the `Happens` instance, choose if the instance is negated, change the value of time if needed (default value is 0), define the argument(s) with the necessary InterRefs and choose if the arguments will be displayed on the drawing area.

Figure 5.13 displays the second chapter of the `Happens` class Notebook. In this chapter, the user can define the constraints that need to be applied in order for an effect to have the desired results. In order to define a constraint, we must choose the type of the constraint (the types are shown in Fig. 5.4), what role the 2 arguments of the constraint have and then define the InterRefs to the objects that include the arguments that are part of the constraint. In our example of the trigger axiom, the constraint we have added is for the 2 instances of type person to be different, `PERSON1 != PERSON2`. Those arguments are part of the triggering event and one of the fluent preconditions (`newPedestrian & waitingP`) so we define the InterRefs to those two objects. Then we choose the role these arguments have, `person` in our case, and then choose the operator from the drop-down list.


Happens-80805 (Happens)

Constraint argument pair:
Happens-80802 Traffic Lights Domain Axiomatization
HoldsAt-80809 Traffic Lights Domain Axiomatization

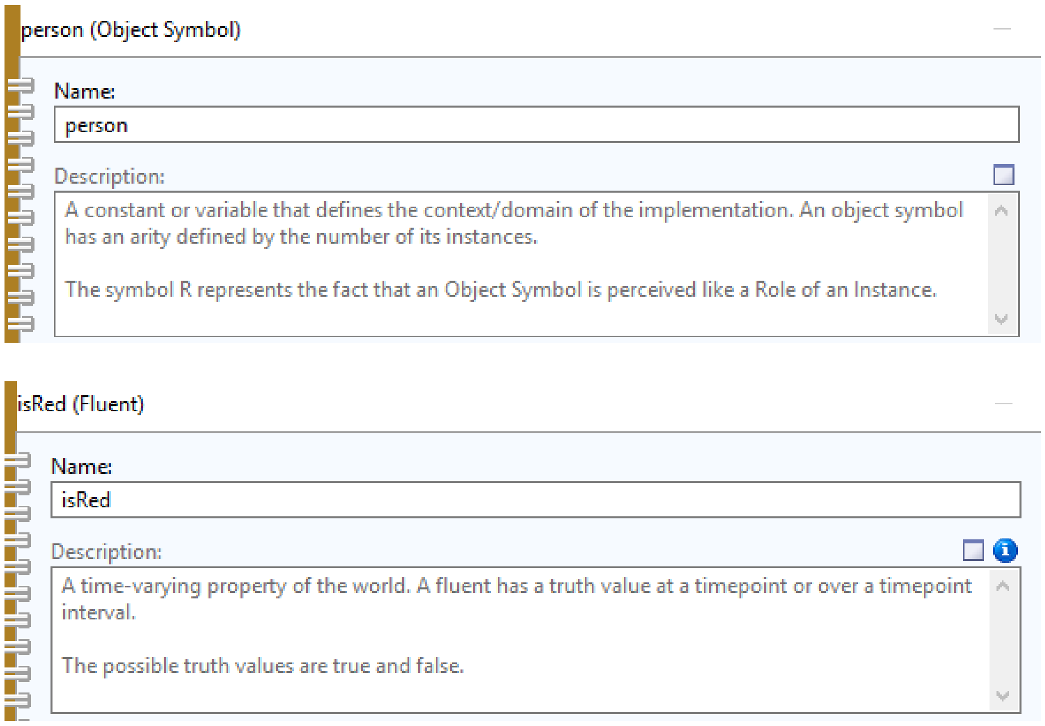
Constraint argument type:
person Traffic Lights Domain

Constraint operator:
!=

Figure 5.13: Example of defining constraints for a triggered event

Notice that, the names of the `Happens` and `HoldsAt` objects are auto-generated when the object is created. The auto-generated name is displayed on the first chapter of the object's notebook, in read-only mode, so that the user will know the name of the object when defining an InterRef to it.

The third chapter of the `Happens` class Notebook includes 2 textfields, *Description* and *Comment*. Those two textfields are part of every object class implemented in ADOxx. We use the Description field in order to provide a sample description of what objects of this class are used for (e.g. what we use an object of type Happens for). The Comment field is for the user to write whatever comment he may find helpful regarding this object instance, during his design. A couple of examples of the Description chapter are shown in Figure 5.14



person (Object Symbol)

Name:
person

Description:
A constant or variable that defines the context/domain of the implementation. An object symbol has an arity defined by the number of its instances.

The symbol R represents the fact that an Object Symbol is perceived like a Role of an Instance.

isRed (Fluent)

Name:
isRed

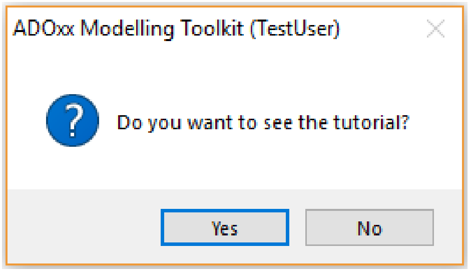
Description:
A time-varying property of the world. A fluent has a truth value at a timepoint or over a timepoint interval.

The possible truth values are true and false.

Figure 5.14: Examples of descriptions of object classes.

All of the above figures are parts of the 4 model types that define our problem (further described in Section 4.2). The user is free to implement the models in any order he desires but in order to make the desired mappings (InterRefs) to other models, a certain order of implementation has to be applied. For this purpose, we designed a walk-through wizard, in the form of infoboxes, that help first time users of the tool learn and understand the correct order to design their domains.

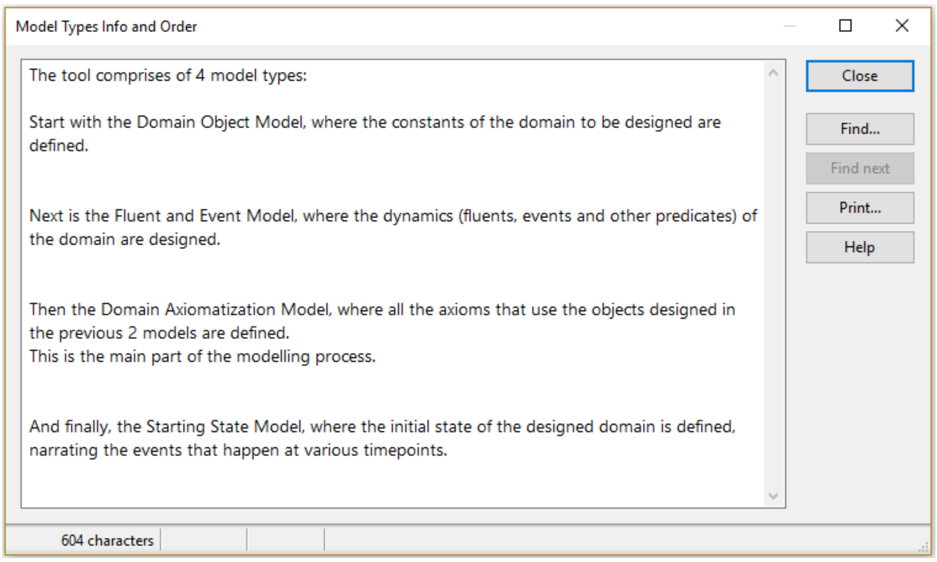
Upon the start of the Modelling Toolkit, the user is asked if he wants the tutorial/infoboxes displayed (Fig. 5.15)


ADOxx Modelling Toolkit (TestUser)

Do you want to see the tutorial?

Figure 5.15: Tutorial: User is asked if he wants the tutorial displayed.

If the "Yes" option is chosen then the tutorial continues, otherwise it is skipped. Upon the click of the "Yes" option a small description of the 4 model types is displayed (Fig. 5.16).


Model Types Info and Order

The tool comprises of 4 model types:

Start with the Domain Object Model, where the constants of the domain to be designed are defined.

Next is the Fluent and Event Model, where the dynamics (fluents, events and other predicates) of the domain are designed.

Then the Domain Axiomatization Model, where all the axioms that use the objects designed in the previous 2 models are defined.
This is the main part of the modelling process.

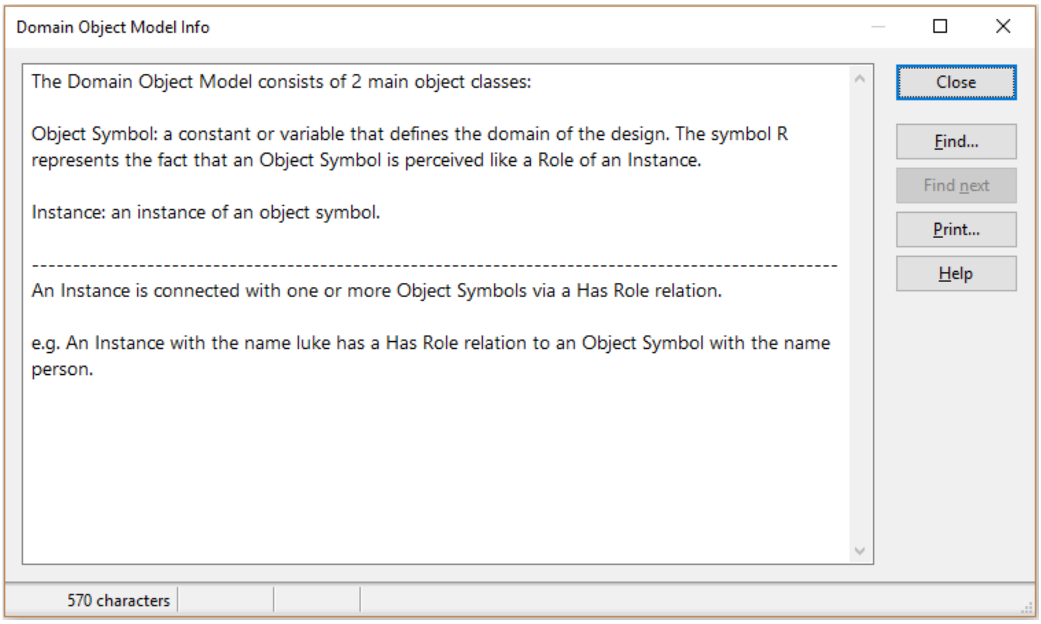
And finally, the Starting State Model, where the initial state of the designed domain is defined, narrating the events that happen at various timepoints.

604 characters

Figure 5.16: Tutorial: Description of the 4 modeltypes

First, the *Domain Object Model* has to be designed for the initialization of the constants that define the problem. Then the *Fluents and Events Model* is designed, after that the *Domain Axiomatization Model* and last is the *Starting State Model*. The last three of those models need InterRefs to the Domain Object Model in order to define the arguments of each predicate and the last 2 models need InterRefs to the Fluents and Events Model, as well, so that fluents and events can be mapped to `HoldsAt` and `Happens` objects correspondingly. This order of the model design also incorporates the semantic of Event Calculus and ASP where the constants and the preconditions must already be defined for the axioms to apply.

Following the above, when each of the 4 modeltypes is opened (i.e. when the model window is activated), a description of the model and the object classes that comprise that model is displayed (Figures 5.17, 5.18, 5.19, 5.20).



The Domain Object Model consists of 2 main object classes:

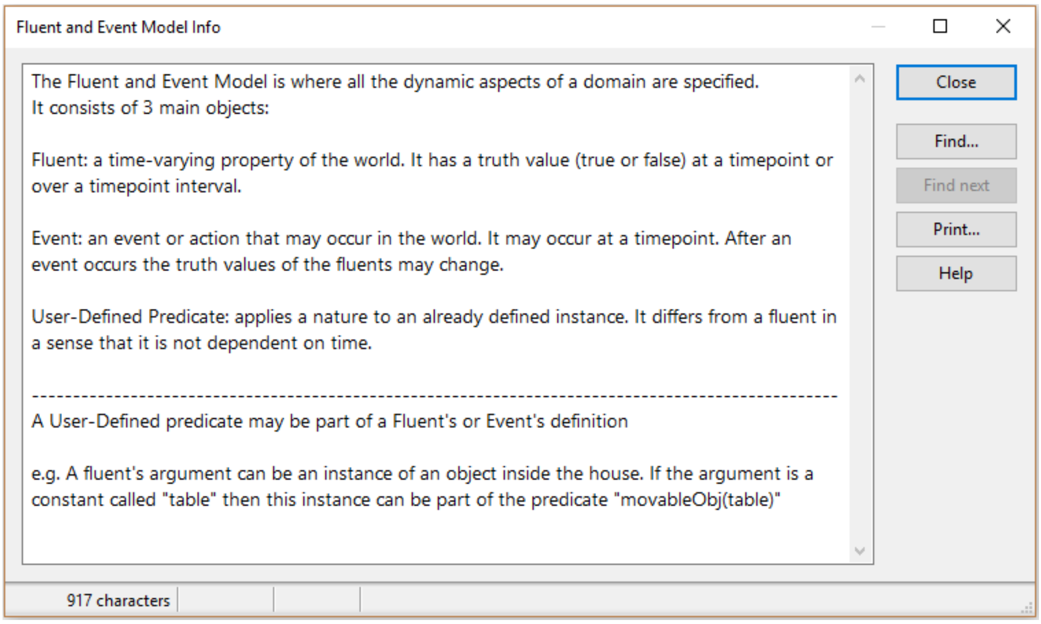
Object Symbol: a constant or variable that defines the domain of the design. The symbol R represents the fact that an Object Symbol is perceived like a Role of an Instance.

Instance: an instance of an object symbol.

An Instance is connected with one or more Object Symbols via a Has Role relation.

e.g. An Instance with the name luke has a Has Role relation to an Object Symbol with the name person.

Figure 5.17: Tutorial: Description displayed for the Domain Object Modeltype



The Fluent and Event Model is where all the dynamic aspects of a domain are specified.
It consists of 3 main objects:

Fluent: a time-varying property of the world. It has a truth value (true or false) at a timepoint or over a timepoint interval.

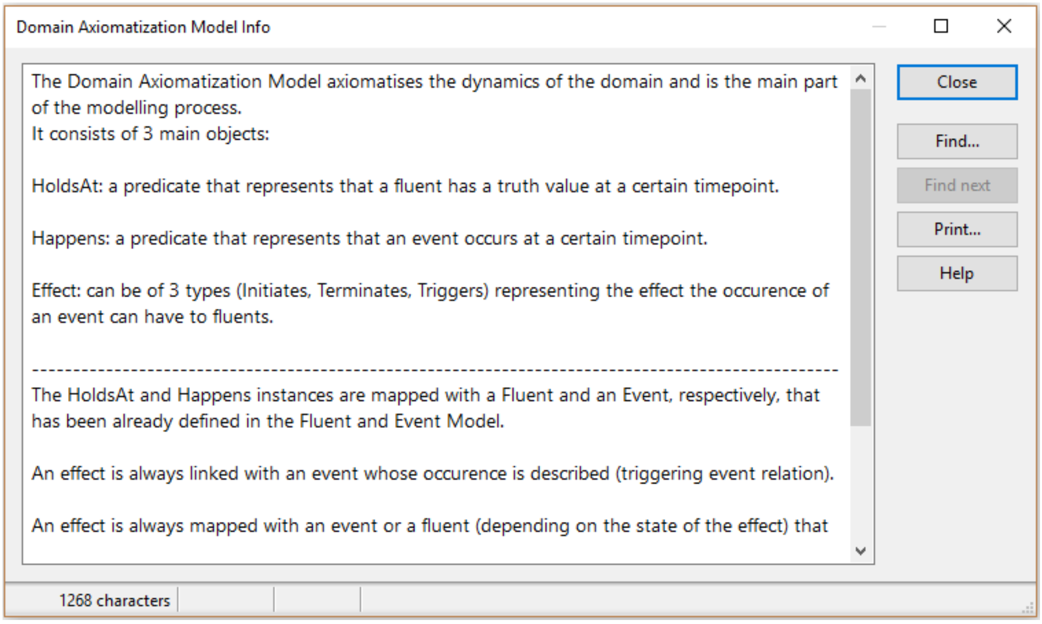
Event: an event or action that may occur in the world. It may occur at a timepoint. After an event occurs the truth values of the fluents may change.

User-Defined Predicate: applies a nature to an already defined instance. It differs from a fluent in a sense that it is not dependent on time.

A User-Defined predicate may be part of a Fluent's or Event's definition

e.g. A fluent's argument can be an instance of an object inside the house. If the argument is a constant called "table" then this instance can be part of the predicate "movableObj(table)"

Figure 5.18: Tutorial: Description displayed for the Fluent and Event Modeltype



The Domain Axiomatization Model axiomatises the dynamics of the domain and is the main part of the modelling process.
It consists of 3 main objects:

HoldsAt: a predicate that represents that a fluent has a truth value at a certain timepoint.

Happens: a predicate that represents that an event occurs at a certain timepoint.

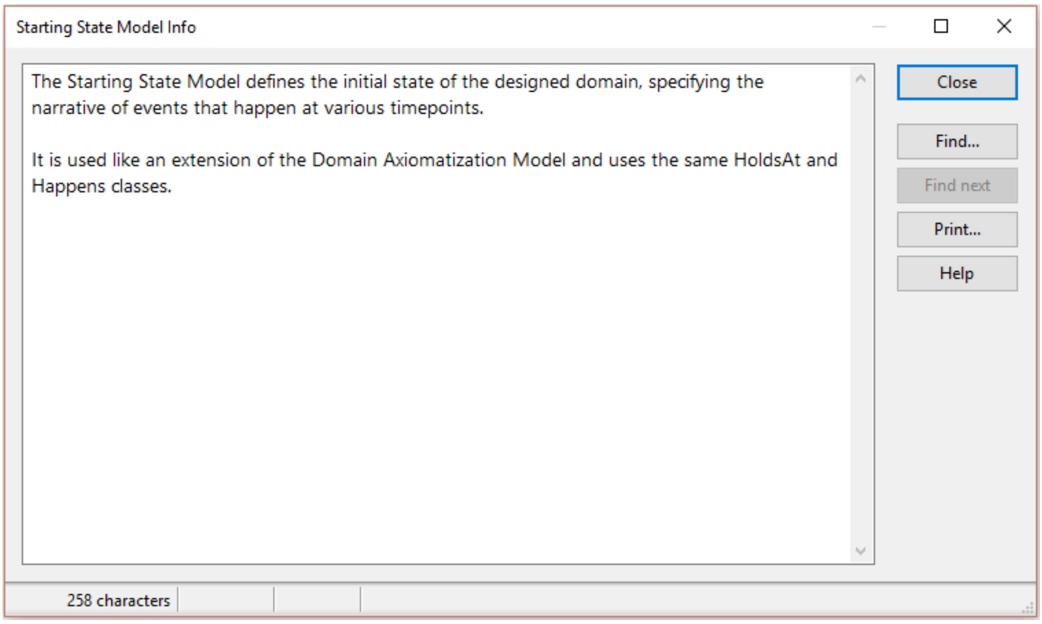
Effect: can be of 3 types (Initiates, Terminates, Triggers) representing the effect the occurence of an event can have to fluents.

The HoldsAt and Happens instances are mapped with a Fluent and an Event, respectively, that has been already defined in the Fluent and Event Model.

An effect is always linked with an event whose occurence is described (triggering event relation).

An effect is always mapped with an event or a fluent (depending on the state of the effect) that

Figure 5.19: Tutorial: Description displayed for the Domain Axiomatization Modeltype

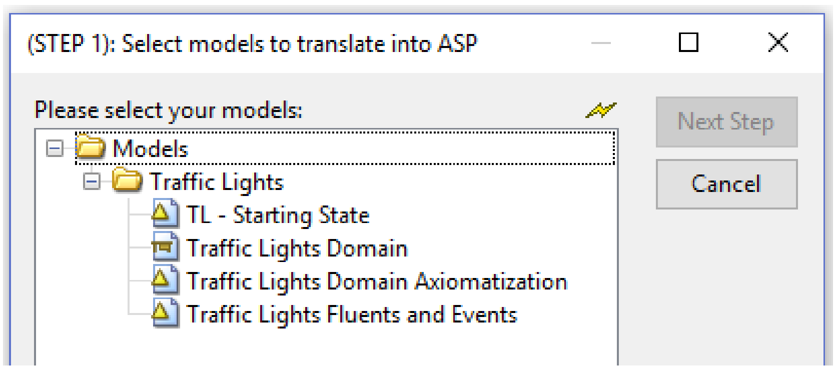


The Starting State Model defines the initial state of the designed domain, specifying the narrative of events that happen at various timepoints.

It is used like an extension of the Domain Axiomatization Model and uses the same HoldsAt and Happens classes.

Figure 5.20: Tutorial: Description displayed for the Starting State Modeltype

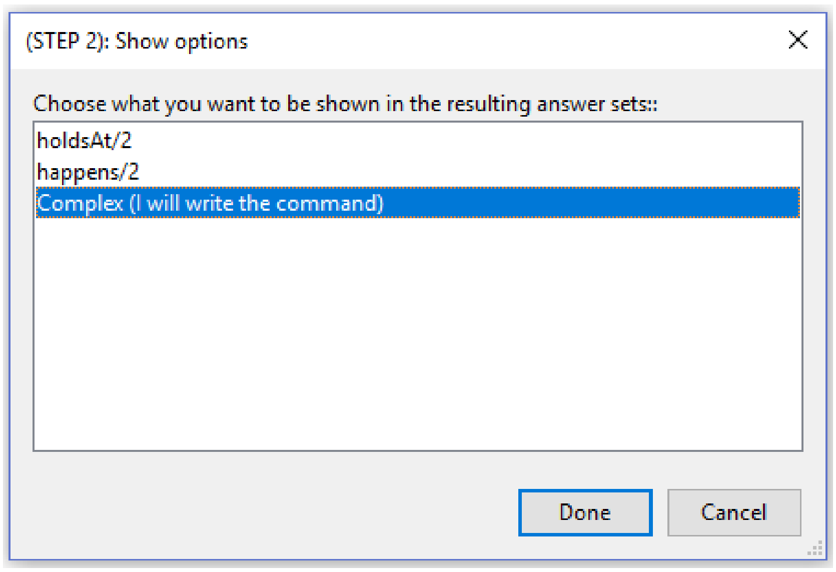
Each of those tooltips are also accessible for the user to read - if need be - later via a top-level menu option called "Show model description".


(STEP 1): Select models to translate into ASP

Please select your models:

- Models
  - Traffic Lights
    - TL - Starting State
    - Traffic Lights Domain
    - Traffic Lights Domain Axiomatization
    - Traffic Lights Fluents and Events

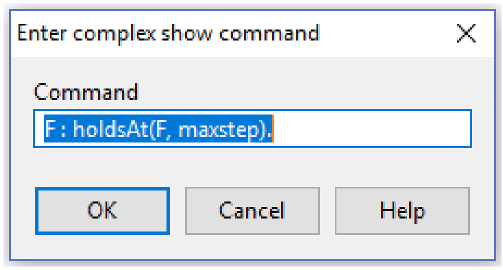
Figure 5.21: Run Step 1: Choose the models to translate


(STEP 2): Show options

Choose what you want to be shown in the resulting answer sets::

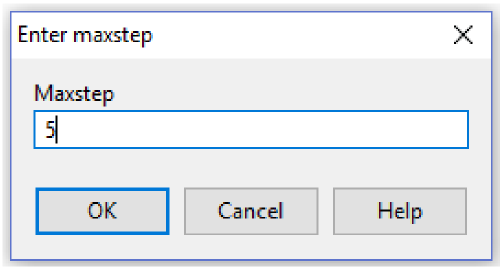
holdsAt/2
happens/2
Complex (I will write the command)

Figure 5.22: Run Step 2: Show option for answer sets


Enter complex show command

Command: F : holdsAt(F, maxstep).

Figure 5.23: Run Step (optional): Define complex show option


Enter maxstep

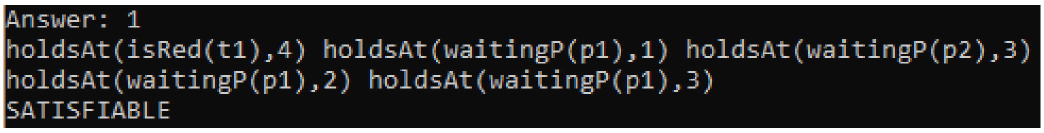
Maxstep: 5

Figure 5.24: Run Step 3: Define maxstep

Figures 5.21, 5.22, 5.23, 5.24 show the steps that follow after the user chooses to run the script that implements the translation to ASP (script 6).

On the first step, the user is called to choose the which models will comprise his ASP program. A check has to be done at this point, if at least one model of each type is chosen since all modeltypes are needed for a complete ASP program. However, more models of each modeltype can be chosen, for if the domain that was designed is too big then it is a good practice to split the design of each modeltype into more than one models.

On the second step, the user chooses the value of the `#show` directive. This directive is used for advising the solver to project stable models onto instances of a certain predicate, meaning if the option `holdsAt/2` is chosen, then for each answer set only the holdsAt instances are going to be displayed. Below an example of an answer set that clingo produces for our traffic lights problem when the show option is holdsAt.



```
Answer: 1
holdsAt(isRed(t1),4) holdsAt(waitingP(p1),1) holdsAt(waitingP(p2),3)
holdsAt(waitingP(p1),2) holdsAt(waitingP(p1),3)
SATISFIABLE
```

Figure 5.25: Example of an answer set produced by clingo

If the user desires to enter a more complex option to the show directive, then he has to choose the last choice shown in Step 2 (Fig. 5.22). When this choice is selected, then the intermediate step shown in Figure 5.23 is displayed and the user can write the command he desires. The example command shown in the figure enables the printing of only certain arguments from a predicate.

The DEC.lp file defines the Discrete Event Calculus Domain independent axiomatization that we use for our program. In this file, the variable `maxstep` is used for defining the max timepoint for which the clingo reasoner is gonna run. On the final step, the user defines the value of the variable maxstep (Fig. 5.24). In DEC.lp we defined:

```
time(0..maxstep)
```

meaning that the variable of time takes values from 0 to maxstep. In our example the maxstep equals to 5 and in the answer set example in Figure 5.25 we can see that the timepoints in the answers are up to the number 5.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we presented ECAVI, a new, domain independent modelling tool, which supports the modelling of dynamic domains in the Event Calculus through a visual interface for generating axioms. In essence, our tool offers a visual meta-modelling platform for knowledge engineering, providing meta-reasoning capabilities to support the inexperienced modeller become acquainted with the features of Event Calculus, and simplifying the process of creating complex models, thereby assisting him/her during the complex learning process.

Visualisation has been identified as a useful means to improve the performance of knowledge engineers, while the need to visualise logical concepts, especially in dynamic domains, has been around for many years; nevertheless, very few tools have focused on the visualisation of action languages. An initial attempt to illustrate similar calculi in an abstract level (like the Situation Calculus) has been presented in [34]; our proposal is more expressive, and provides a more distinguishable flow of actions. The SeaLion IDE for Answer-Set Programming [28] is also making a remarkable attempt towards providing graphical representation for the ASP language, yet it only focuses on a specific target group of users with some or more extensive background knowledge. Moreover, VizDSL [26], a visual Domain-Specific Language for highly interactive visualisations argues that "visualisation facilitates knowledge sharing between different user groups with different levels of expertise and experience, without requiring extensive background knowledge or training". Similar to our work, [4] introduced an interactive educational tool for students that want to learn AI planning for robotics. The tool guides the student to model a dynamic domain and solve a planning problem, however no graphical visualisation of the axioms is provided. ECAVI is an attempt to offer a complete solution towards this direction, being a domain-independent educational tool that models dynamic domains via a visual interface that focuses on making the student understand actions, change and causality. The Unified Modelling Language (UML) [9] has been a guide towards realising a visual notation for our purposes.

6.2 Future Work

ECAVI is in a work-in-progress phase and is only a first prototype. The features presented in Chapter 5 have already proven their usefulness in some preliminary tests, but cannot be considered complete, and do not fully realise our vision towards a meta-modelling platform for modelling dynamic domains. Despite the encouraging (informal) feedback received by students who tried the tool, we still need to perform a more extensive normative evaluation of the current features of ECAVI that will quantify the gains in terms of modelling time and task completion time that users of different levels have while using the tool (as opposed to the standard baseline of using a plain text editor). Specifically, we will evaluate the tool's usability in terms of *satisfaction, efficiency* and *effectiveness* by means of user studies both from the perspective of the variety of students with different backgrounds, as well as, the perspective of the teacher.

We already identified some bugs in the modelling interface what will be finished in a subsequent step. The need for minor revisions of the current features and/or the incorporation of new ones is expected to emerge through the evaluation process, allowing us to further improve and refine the usability of the tool and to identify weak/strong features.

Moreover, upcoming versions will consider more features of the Event Calculus, such as non-determinism, indirect effects of events and others. With the integration of more capabilities in our tool, full programs that have already been implemented in ASP can be imported and visualised with the help of ECAVI. We also plan extend the integrity checks and the meta-reasoning capabilities of our implementation, by raising warnings and exceptions during design time.

Furthermore, we plan to extend our focus to other types of users, i.e., users with different levels of modelling experience. Clearly, more experienced users have different needs, therefore new features will have to be supported, such as more complex meta-reasoning tools and tools analysing the resulting models to identify potential errors or poor modelling choices. Such features may be useful to experienced users who typically model large domains with hundreds of rules.

In the long run, we envision ECAVI to take the form of a fully visual integrated development environment for modelling dynamic domains, complete with a debugger, step-by-step execution and other features typically found in IDEs, while supporting alternative action languages, such as the Situation Calculus [24, 22] or similar action formalisms.

Bibliography

[1] Alexander Artikis, Marek Sergot, and Jeremy Pitt. An executable specification of a formal argumentation protocol. *Artif. Intell.*, 171(10-15):776–804, July 2007.

[2] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice: Second Edition*. Morgan & Claypool Publishers, 2nd edition, 2017.

[3] Alberto Rodrigues da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139–155, 2015.

[4] Zeynep Dogmus, Esra Erdem, and Volkan Patoglu. React!: An interactive educational tool for ai planning for robotics. 58(1):15–24, 2015. Exported from https://app.dimensions.ai on 2018/11/13.

[5] Uwe Egly, Sarah Alice Gaggl, and Stefan Woltran. Aspartix: Implementing argumentation frameworks using answer-set programming. In *Proceedings of the 24th International Conference on Logic Programming*, ICLP '08, pages 734–738, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] Wolfgang Faber and Stefan Woltran. Manifold answer-set programs for meta-reasoning. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning*, LPNMR '09, pages 115–128, Berlin, Heidelberg, 2009. Springer-Verlag.

[7] Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. Stable models and circumscription. *Artificial Intelligence*, 175(1):236–263, 2011.

[8] Hans-Georg Fill and Dimitris Karagiannis. On the conceptualisation of modelling methods using the ADOxx meta modelling platform. *Enterprise Modelling and Information Systems Architectures*, 8:4–25, 2013.

[9] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003.

[10] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Clasp: A conflict-driven answer set solver. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning*, LPNMR'07, pages 260–265, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. pages 1070–1080. MIT Press, 1988.

[12] David Götzinger, Elena-Teodora Miron, and Franz Staffel. Omilab: An open collaborative environment for modeling method engineering. 2016.

[13] Knut Hinkelmann. *Business Process Flexibility and Decision-Aware Modeling—The Knowledge Work Designer*, pages 397–414. Springer International Publishing, Cham, 2016.

[14] Dimitris Karagiannis. Agile modeling method engineering. In *Proceedings of the 19th Panhellenic Conference on Informatics*, PCI '15, pages 5–10, New York, NY, USA, 2015. ACM.

[15] Dimitris Karagiannis, Robert Andrei Buchmann, Patrik Burzynski, Ulrich Reimer, and Michael Walch. Fundamental conceptual modeling languages in omilab. In *Domain-Specific Conceptual Modeling*, 2016.

[16] Dimitris Karagiannis and Harald Kühn. Metamodelling platforms. In *Proceedings of the Third International Conference on E-Commerce and Web Technologies*, EC-WEB '02, pages 182–, London, UK, UK, 2002. Springer-Verlag.

[17] Dimitris Karagiannis, Heinrich C. Mayr, and John Mylopoulos. *Domain-Specific Conceptual Modeling: Concepts, Methods and Tools.* Springer Publishing Company, Incorporated, 1st edition, 2016.

[18] Christian Kloimüllner, Johannes Oetsch, Jörg Pührer, and Hans Tompits. Kara: A system for visualising and visual editing of interpretations for answer-set programs. In Hans Tompits, Salvador Abreu, Johannes Oetsch, Jörg Pührer, Dietmar Seipel, Masanobu Umeda, and Armin Wolf, editors, *Applications of Declarative Programming and Knowledge Management*, pages 325–344, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[19] RA Kowalski and MJ Sergot. A logic-based calculus of events. newgeneration computing 4., 1986.

[20] Frank Kramer and Bernhard Thalheim. Holistic conceptual and logical database structure modeling with adoxx. In *Domain-Specific Conceptual Modeling*, 2016.

[21] Joohyung Lee and Ravi Palla. Reformulating the Situation Calculus and the Event Calculus in the General Theory of Stable Models and in Answer Set Programming. *Journal of Artificial Intelligence Research*, 43(1):571–620, January 2012.

[22] Hector J. Levesque, Fiora Pirri, and Raymond Reiter. Foundations for the situation calculus. *Electron. Trans. Artif. Intell.*, 2:159–178, 1998.

[23] Vladimir Lifschitz. What is answer set programming? In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, AAAI'08, pages 1594–1597. AAAI Press, 2008.

[24] J. McCarthy and Stanford Artificial Intelligence Laboratory. *Situations, Actions, and Causal Laws.* Memo (Stanford Artificial Intelligence Project). Comtex Scientific, 1963.

[25] Rob Miller and Murray Shanahan. Some alternative formulations of the event calculus. In *Computational logic: logic programming and beyond*, pages 452–490. Springer, 2002.

[26] Rebecca Morgan, Georg Grossmann, and Markus Stumptner. Vizdsl: Towards a graphical visualisation language for enterprise systems interoperability. *2017 International Symposium on Big Data Visual Analytics (BDVA)*, pages 1–8, 2017.

[27] Erik Mueller. *Commonsense Reasoning.* Morgan Kaufmann, 1st edition, 2006.

[28] Johannes Oetsch, Jörg Pührer, and Hans Tompits. The sealion has landed: An ide for answer-set programming—preliminary report. In Hans Tompits, Salvador Abreu, Johannes Oetsch, Jörg Pührer, Dietmar Seipel, Masanobu Umeda, and Armin Wolf, editors, *Applications of Declarative Programming and Knowledge Management*, pages 305–324, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[29] Tony Ribeiro, Katsumi Inoue, and Gauvain Bourgne. Combining Answer Set Programs for Adaptive and Reactive Reasoning. *Theory and Practice of Logic Programming*, 13(4-5-Online-Supplement), July 2013.

[30] Mohsen Rouached, Olivier Perrin, and Claude Godart. Retracted: Towards formal verification of web service composition. In Schahram Dustdar, José Luiz Fiadeiro, and Amit P. Sheth, editors, *Business Process Management*, pages 257–273, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[31] F. Van Harmelen, V. Lifschitz, and B. Porter. *Handbook of Knowledge Representation.* Elsevier Science, San Diego, USA, 2007.

[32] Michiel van Lambalgen and Fritz Hamm. The proper treatment of events. *Erkenntnis*, 65(3):441–447, 2006.

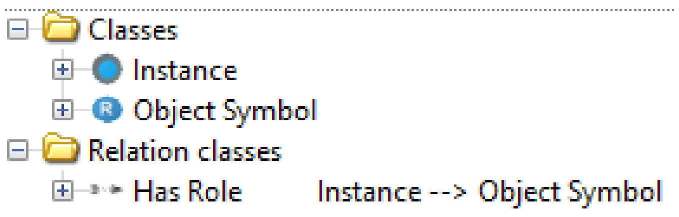
[33] Niksa Visic, Hans-Georg Fill, Robert Andrei Buchmann, and Dimitris Karagiannis. A domain-specific language for modeling method definition: From requirements to grammar. *2015 IEEE 9th International Conference on Research Challenges in Information Science (RCIS)*, pages 286–297, 2015.

[34] Susumu Yamasaki and Mariko Sasakura. A calculus effectively performing event formation with visualization. In Jesús Labarta, Kazuki Joe, and Toshinori Sato, editors, *High-Performance Computing*, pages 287–294, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

Appendices

Appendix A

List of Classes & Relation Classes

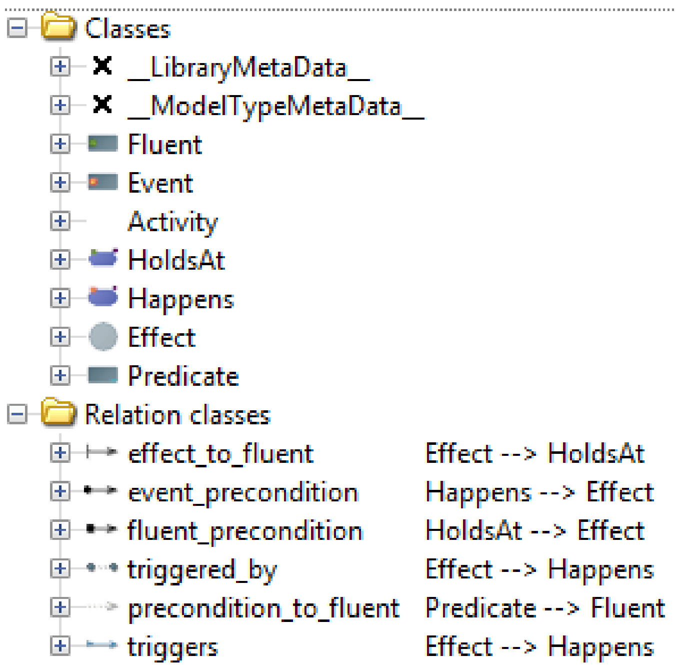


Classes
- Instance
- Object Symbol

Relation classes

Has Role	Instance --> Object Symbol

Figure A.1: List of classes and relation classes of the ADOxx Static Library



Classes
- __LibraryMetaData__
- __ModelTypeMetaData__
- Fluent
- Event
- Activity
- HoldsAt
- Happens
- Effect
- Predicate

Relation classes

effect_to_fluent	Effect --> HoldsAt
event_precondition	Happens --> Effect
fluent_precondition	HoldsAt --> Effect
triggered_by	Effect --> Happens
precondition_to_fluent	Predicate --> Fluent
triggers	Effect --> Happens

Figure A.2: List of classes and relation classes of the ADOxx Dynamic Library

Appendix B

Sample List of Scripts in AdoScript

Algorithm 4 Auto-generate Triggering Event (Happens)

```
    # Get the "Type" attribute value of from object
    CC "Core"  GET_ATTR_VAL objid: (idInstId) attrname: ("Type")
    SETL type: (val)

    IF (type="Initiates" OR type="Terminates" OR type="Triggers") {
        # get the id of class "Happens"
        CC "Core" GET_CLASS_ID classname:"Happens"

        # create the object/instance of the class
        CC "Core" CREATE_OBJ modelid:(idModelId) classid:(classid)
        IF (ecode != 0) {
            CC "AdoScript" ERRORBOX ("The object could not be
                created!")
        }
        SETL newInstanceID:(objid)

        # get the id of the connector "triggered_by"
        CC "Core" GET_CLASS_ID relation classname:"triggered_by"
        IF (ecode != 0) {
            CC "AdoScript" ERRORBOX ("The relation triggered_by
                couldn't be found!")
        }
        CC "Core" CREATE_CONNECTOR modelid:(idModelId)
            classid:(classid)
            fromobjid:(instid) toobjid:(newInstanceID)
        IF (ecode != 0) {
            CC "AdoScript" ERRORBOX ("The relation triggered_by
                could not be created!")
        }
        # this has to be called to update the modeling window
        CC "Modeling" REBUILD_DRAWING_AREA
    }
}
```

Algorithm 5 Auto-generate Preconditions

```
# get the id of class "HoldsAt"
CC "Core" GET_CLASS_ID classname:"HoldsAt"

# create the object/instance of the class
CC "Core" CREATE_OBJ modelid:(id_ModelId) classid:(classid)
IF (ecode != 0) {
    CC "AdoScript" ERRORBOX ("The object could not be created!")
}

SETL newInstanceID:(objid)

# get the id of the connector "fluent_precondition"
CC "Core" GET_CLASS_ID relation classname:"fluent_precondition"
IF (ecode != 0) {
    CC "AdoScript" ERRORBOX ("The relation fluent_precondition
        couldn't be found!")
}

CC "Core" CREATE_CONNECTOR modelid:(id_ModelId) classid:(classid)
                    fromobjid:(newInstanceID) toobjid:(id_InstId)
IF (ecode != 0) {
    CC "AdoScript" ERRORBOX ("The relation triggered_by could not be
         created!")
}

# this has to be called to update the modeling window
CC "Modeling" REBUILD_DRAWING_AREA
```

Algorithm 6 Run Export and Translation (only key parts)

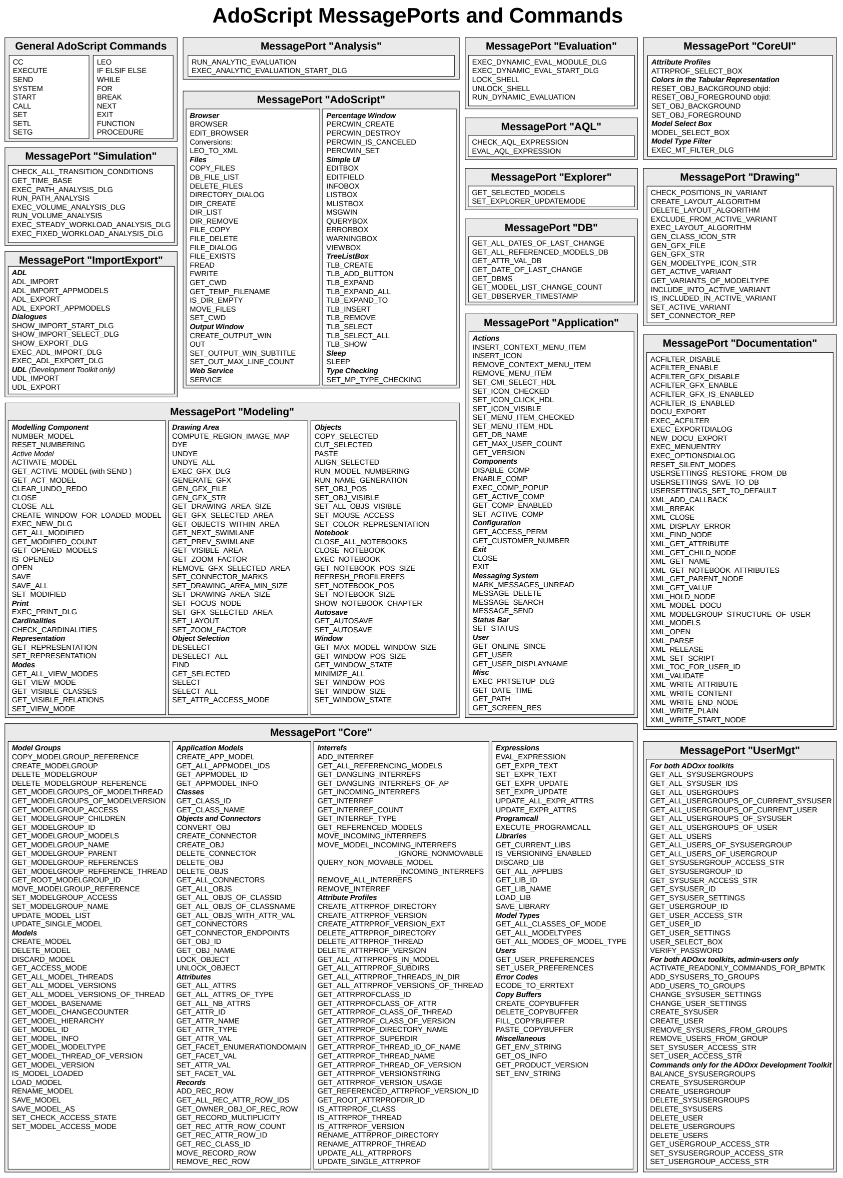
```
CC "Modeling" GET_ACT_MODEL
SETL nCurrentModelID: (modelid)
IF (modelid != -1) {
    CC "Core" GET_MODEL_INFO modelid: (nCurrentModelID)
    SETL sCurrentModeltype: (modeltype)

    # only run script if a model of a specific type is open
    IF (sCurrentModeltype = "Starting state model") {
        # Step 2 - open a configuration screen.
        # a configuration screen can be realized using the CoreUI
            MessagePort for model select boxes and the like or custom
             screens using a DLL for the matrix dialog
        CC "CoreUI" MODEL_SELECT_BOX
            oktext:"Next Step" boxtext:"Please select your models:"
            title:"(STEP 1): Select models to translate into ASP"
            multi-sel mgroup-sel setdbclick:0
        SETL boxResult: (endbutton)
        ...
        # Export model as XML
        ...
        # Write export to tmp file
        SET sJARpath:   ("<insert JAR path here>")
        SET sTranslationResult:
                        ("<local path>\\translatedFromXML.lp")
        ...
        # perform a transformation using external functionality
            developed in Java
        SYSTEM ("cmd /c java -jar " + sJARpath + " " + sJARargs)
            result:rc
        IF (rc != 0) {
            CC "AdoScript" ERRORBOX ("File with name: " +
                sXMLFileName + " didn't run.")
            EXIT
        }
        # set file were the Clingo results will be printed into
        SET sClingoResult: ("<local path>\\TL_clingo_result.txt\"")

        SYSTEM ("cmd /c clingo.exe <local path>\\DEC.lp \"" +
            sTranslationResult + "\" -c maxstep=" + maxstep + " > " +
             sClingoResult) result:rc
        IF (rc = 0) {
            CC "AdoScript" ERRORBOX ("Couldn't run clingo!")
            EXIT
        }
        CC "AdoScript" INFOBOX ("Run Complete!")
    }
}
```

Appendix C

AdoScript MessagePorts and Commands



AdoScript MessagePorts and Commands

General AdoScript Commands

- CC
- EXECUTE
- SEND
- SYSTEM
- START
- CALL
- SET
- SETL
- SETG

- LEO
- IF ELSIF ELSE
- WHILE
- FOR
- BREAK
- NEXT
- EXIT
- FUNCTION
- PROCEDURE

MessagePort "Simulation"

- CHECK_ALL_TRANSITION_CONDITIONS
- GET_TIME_BASE
- EXEC_PATH_ANALYSIS_DLG
- RUN_PATH_ANALYSIS
- EXEC_VOLUME_ANALYSIS_DLG
- RUN_VOLUME_ANALYSIS
- EXEC_STEADY_WORKLOAD_ANALYSIS_DLG
- EXEC_FIXED_WORKLOAD_ANALYSIS_DLG

MessagePort "ImportExport"

- ***ADL***
- ADL_IMPORT
- ADL_IMPORT_APPMODELS
- ADL_EXPORT
- ADL_EXPORT_APPMODELS
- ***Dialogues***
- SHOW_IMPORT_START_DLG
- SHOW_IMPORT_SELECT_DLG
- SHOW_EXPORT_DLG
- EXEC_ADL_IMPORT_DLG
- EXEC_ADL_EXPORT_DLG
- ***UDL*** *(Development Toolkit only)*
- UDL_IMPORT
- UDL_EXPORT

MessagePort "Analysis"

- RUN_ANALYTIC_EVALUATION
- EXEC_ANALYTIC_EVALUATION_START_DLG

MessagePort "AdoScript"

- ***Browser***
- BROWSER
- EDIT_BROWSER
- Conversions:
- LEO_TO_XML
- ***Files***
- COPY_FILES
- DB_FILE_LIST
- DELETE_FILES
- DIRECTORY_DIALOG
- DIR_CREATE
- DIR_LIST
- DIR_REMOVE
- FILE_COPY
- FILE_DELETE
- FILE_DIALOG
- FILE_EXISTS
- FREAD
- FWRITE
- GET_CWD
- GET_TEMP_FILENAME
- IS_DIR_EMPTY
- MOVE_FILES
- SET_CWD
- ***Output Window***
- CREATE_OUTPUT_WIN
- OUT
- SET_OUTPUT_WIN_SUBTITLE
- SET_OUT_MAX_LINE_COUNT
- ***Web Service***
- SERVICE

- ***Percentage Window***
- PERCWIN_CREATE
- PERCWIN_DESTROY
- PERCWIN_IS_CANCELED
- PERCWIN_SET
- ***Simple UI***
- EDITBOX
- EDITFIELD
- INFOBOX
- LISTBOX
- MLISTBOX
- MSGWIN
- QUERYBOX
- ERRORBOX
- WARNINGBOX
- VIEWBOX
- ***TreeListBox***
- TLB_CREATE
- TLB_ADD_BUTTON
- TLB_EXPAND
- TLB_EXPAND_ALL
- TLB_EXPAND_TO
- TLB_INSERT
- TLB_REMOVE
- TLB_SELECT
- TLB_SELECT_ALL
- TLB_SHOW
- ***Sleep***
- SLEEP
- ***Type Checking***
- SET_MP_TYPE_CHECKING

MessagePort "Evaluation"

- EXEC_DYNAMIC_EVAL_MODULE_DLG
- EXEC_DYNAMIC_EVAL_START_DLG
- LOCK_SHELL
- UNLOCK_SHELL
- RUN_DYNAMIC_EVALUATION

MessagePort "AQL"

- CHECK_AQL_EXPRESSION
- EVAL_AQL_EXPRESSION

MessagePort "Explorer"

- GET_SELECTED_MODELS
- SET_EXPLORER_UPDATEMODE

MessagePort "DB"

- GET_ALL_DATES_OF_LAST_CHANGE
- GET_ALL_REFERENCED_MODELS_DB
- GET_ATTR_VAL_DB
- GET_DATE_OF_LAST_CHANGE
- GET_DBMS
- GET_MODEL_LIST_CHANGE_COUNT
- GET_DBSERVER_TIMESTAMP

MessagePort "CoreUI"

- ***Attribute Profiles***
- ATTRPROF_SELECT_BOX
- ***Colors in the Tabular Representation***
- RESET_OBJ_BACKGROUND objid:
- RESET_OBJ_FOREGROUND objid:
- SET_OBJ_BACKGROUND
- SET_OBJ_FOREGROUND
- ***Model Select Box***
- MODEL_SELECT_BOX
- ***Model Type Filter***
- EXEC_MT_FILTER_DLG

MessagePort "Drawing"

- CHECK_POSITIONS_IN_VARIANT
- CREATE_LAYOUT_ALGORITHM
- DELETE_LAYOUT_ALGORITHM
- EXCLUDE_FROM_ACTIVE_VARIANT
- EXEC_LAYOUT_ALGORITHM
- GEN_CLASS_ICON_STR
- GEN_GFX_FILE
- GEN_GFX_STR
- GEN_MODELTYPE_ICON_STR
- GET_ACTIVE_VARIANT
- GET_VARIANTS_OF_MODELTYPE
- INCLUDE_INTO_ACTIVE_VARIANT
- IS_INCLUDED_IN_ACTIVE_VARIANT
- SET_ACTIVE_VARIANT
- SET_CONNECTOR_REP

MessagePort "Modeling"

- ***Modelling Component***
- NUMBER_MODEL
- RESET_NUMBERING
- *Active Model*
- ACTIVATE_MODEL
- GET_ACTIVE_MODEL (with SEND)
- GET_ACT_MODEL
- CLEAR_UNDO_REDO
- CLOSE
- CLOSE_ALL
- CREATE_WINDOW_FOR_LOADED_MODEL
- EXEC_NEW_DLG
- GET_ALL_MODIFIED
- GET_MODIFIED_COUNT
- GET_OPENED_MODELS
- IS_OPENED
- OPEN
- SAVE
- SAVE_ALL
- SET_MODIFIED
- ***Print***
- EXEC_PRINT_DLG
- ***Cardinalities***
- CHECK_CARDINALITIES
- ***Representation***
- GET_REPRESENTATION
- SET_REPRESENTATION
- ***Modes***
- GET_ALL_VIEW_MODES
- GET_VIEW_MODE
- GET_VISIBLE_CLASSES
- GET_VISIBLE_RELATIONS
- SET_VIEW_MODE

- ***Drawing Area***
- COMPUTE_REGION_IMAGE_MAP
- DYE
- UNDYE
- UNDYE_ALL
- EXEC_GFX_DLG
- GENERATE_GFX
- GEN_GFX_FILE
- GEN_GFX_STR
- GET_DRAWING_AREA_SIZE
- GET_GFX_SELECTED_AREA
- GET_OBJECTS_WITHIN_AREA
- GET_NEXT_SWIMLANE
- GET_PREV_SWIMLANE
- GET_VISIBLE_AREA
- GET_ZOOM_FACTOR
- REMOVE_GFX_SELECTED_AREA
- SET_CONNECTOR_MARKS
- SET_DRAWING_AREA_MIN_SIZE
- SET_DRAWING_AREA_SIZE
- SET_FOCUS_NODE
- SET_GFX_SELECTED_AREA
- SET_LAYOUT
- SET_ZOOM_FACTOR
- ***Object Selection***
- DESELECT
- DESELECT_ALL
- FIND
- GET_SELECTED
- SELECT
- SELECT_ALL
- SET_ATTR_ACCESS_MODE

- ***Objects***
- COPY_SELECTED
- CUT_SELECTED
- PASTE
- ALIGN_SELECTED
- RUN_MODEL_NUMBERING
- RUN_NAME_GENERATION
- SET_OBJ_POS
- SET_OBJ_VISIBLE
- SET_ALL_OBJS_VISIBLE
- SET_MOUSE_ACCESS
- SET_COLOR_REPRESENTATION
- ***Notebook***
- CLOSE_ALL_NOTEBOOKS
- CLOSE_NOTEBOOK
- EXEC_NOTEBOOK
- GET_NOTEBOOK_POS_SIZE
- REFRESH_PROFILEREFS
- SET_NOTEBOOK_POS
- SET_NOTEBOOK_SIZE
- SHOW_NOTEBOOK_CHAPTER
- ***Autosave***
- GET_AUTOSAVE
- SET_AUTOSAVE
- ***Window***
- GET_MAX_MODEL_WINDOW_SIZE
- GET_WINDOW_POS_SIZE
- GET_WINDOW_STATE
- MINIMIZE_ALL
- SET_WINDOW_POS
- SET_WINDOW_SIZE
- SET_WINDOW_STATE

MessagePort "Application"

- ***Actions***
- INSERT_CONTEXT_MENU_ITEM
- INSERT_ICON
- REMOVE_CONTEXT_MENU_ITEM
- REMOVE_MENU_ITEM
- SET_CMI_SELECT_HDL
- SET_ICON_CHECKED
- SET_ICON_CLICK_HDL
- SET_ICON_VISIBLE
- SET_MENU_ITEM_CHECKED
- SET_MENU_ITEM_HDL
- GET_DB_NAME
- GET_MAX_USER_COUNT
- GET_VERSION
- ***Components***
- DISABLE_COMP
- ENABLE_COMP
- EXEC_COMP_POPUP
- GET_ACTIVE_COMP
- GET_COMP_ENABLED
- SET_ACTIVE_COMP
- ***Configuration***
- GET_ACCESS_PERM
- GET_CUSTOMER_NUMBER
- ***Exit***
- CLOSE
- EXIT
- ***Messaging System***
- MARK_MESSAGES_UNREAD
- MESSAGE_DELETE
- MESSAGE_SEARCH
- MESSAGE_SEND
- ***Status Bar***
- SET_STATUS
- ***User***
- GET_ONLINE_SINCE
- GET_USER
- GET_USER_DISPLAYNAME
- ***Misc***
- EXEC_PRTSETUP_DLG
- GET_DATE_TIME
- GET_PATH
- GET_SCREEN_RES

MessagePort "Documentation"

- ACFILTER_DISABLE
- ACFILTER_ENABLE
- ACFILTER_GFX_DISABLE
- ACFILTER_GFX_ENABLE
- ACFILTER_GFX_IS_ENABLED
- ACFILTER_IS_ENABLED
- DOCU_EXPORT
- EXEC_ACFILTER
- EXEC_EXPORTDIALOG
- NEW_DOCU_EXPORT
- EXEC_MENUENTRY
- EXEC_OPTIONSDIALOG
- RESET_SILENT_MODES
- USERSETTINGS_RESTORE_FROM_DB
- USERSETTINGS_SAVE_TO_DB
- USERSETTINGS_SET_TO_DEFAULT
- XML_ADD_CALLBACK
- XML_BREAK
- XML_CLOSE
- XML_DISPLAY_ERROR
- XML_FIND_NODE
- XML_GET_ATTRIBUTE
- XML_GET_CHILD_NODE
- XML_GET_NAME
- XML_GET_NOTEBOOK_ATTRIBUTES
- XML_GET_PARENT_NODE
- XML_GET_VALUE
- XML_HOLD_NODE
- XML_MODEL_DOCU
- XML_MODELGROUP_STRUCTURE_OF_USER
- XML_MODELS
- XML_OPEN
- XML_PARSE
- XML_RELEASE
- XML_SET_SCRIPT
- XML_TOC_FOR_USER_ID
- XML_VALIDATE
- XML_WRITE_ATTRIBUTE
- XML_WRITE_CONTENT
- XML_WRITE_END_NODE
- XML_WRITE_PLAIN
- XML_WRITE_START_NODE

MessagePort "Core"

- ***Model Groups***
- COPY_MODELGROUP_REFERENCE
- CREATE_MODELGROUP
- DELETE_MODELGROUP
- DELETE_MODELGROUP_REFERENCE
- GET_MODELGROUPS_OF_MODELTHREAD
- GET_MODELGROUPS_OF_MODELVERSION
- GET_MODELGROUP_ACCESS
- GET_MODELGROUP_CHILDREN
- GET_MODELGROUP_ID
- GET_MODELGROUP_MODELS
- GET_MODELGROUP_NAME
- GET_MODELGROUP_PARENT
- GET_MODELGROUP_REFERENCES
- GET_MODELGROUP_REFERENCE_THREAD
- GET_ROOT_MODELGROUP_ID
- MOVE_MODELGROUP_REFERENCE
- SET_MODELGROUP_ACCESS
- SET_MODELGROUP_NAME
- UPDATE_MODEL_LIST
- UPDATE_SINGLE_MODEL
- ***Models***
- CREATE_MODEL
- DELETE_MODEL
- DISCARD_MODEL
- GET_ACCESS_MODE
- GET_ALL_MODEL_THREADS
- GET_ALL_MODEL_VERSIONS
- GET_ALL_MODEL_VERSIONS_OF_THREAD
- GET_MODEL_BASENAME
- GET_MODEL_CHANGECOUNTER
- GET_MODEL_HIERARCHY
- GET_MODEL_ID
- GET_MODEL_INFO
- GET_MODEL_MODELTYPE
- GET_MODEL_THREAD_OF_VERSION
- GET_MODEL_VERSION
- IS_MODEL_LOADED
- LOAD_MODEL
- RENAME_MODEL
- SAVE_MODEL
- SAVE_MODEL_AS
- SET_CHECK_ACCESS_STATE
- SET_MODEL_ACCESS_MODE

- ***Application Models***
- CREATE_APP_MODEL
- GET_ALL_APPMODEL_IDS
- GET_APPMODEL_ID
- GET_APPMODEL_INFO
- ***Classes***
- GET_CLASS_ID
- GET_CLASS_NAME
- ***Objects and Connectors***
- CONVERT_OBJ
- CREATE_CONNECTOR
- CREATE_OBJ
- DELETE_CONNECTOR
- DELETE_OBJ
- DELETE_OBJS
- GET_ALL_CONNECTORS
- GET_ALL_OBJS
- GET_ALL_OBJS_OF_CLASSID
- GET_ALL_OBJS_OF_CLASSNAME
- GET_ALL_OBJS_WITH_ATTR_VAL
- GET_CONNECTORS
- GET_CONNECTOR_ENDPOINTS
- GET_OBJ_ID
- GET_OBJ_NAME
- LOCK_OBJECT
- UNLOCK_OBJECT
- ***Attributes***
- GET_ALL_ATTRS
- GET_ALL_ATTRS_OF_TYPE
- GET_ALL_NB_ATTRS
- GET_ATTR_ID
- GET_ATTR_NAME
- GET_ATTR_TYPE
- GET_ATTR_VAL
- GET_FACET_ENUMERATIONDOMAIN
- GET_FACET_VAL
- SET_ATTR_VAL
- SET_FACET_VAL
- ***Records***
- ADD_REC_ROW
- GET_ALL_REC_ATTR_ROW_IDS
- GET_OWNER_OBJ_OF_REC_ROW
- GET_RECORD_MULTIPLICITY
- GET_REC_ATTR_ROW_COUNT
- GET_REC_ATTR_ROW_ID
- GET_REC_CLASS_ID
- MOVE_RECORD_ROW
- REMOVE_REC_ROW

- ***Interrefs***
- ADD_INTERREF
- GET_ALL_REFERENCING_MODELS
- GET_DANGLING_INTERREFS
- GET_DANGLING_INTERREFS_OF_AP
- GET_INCOMING_INTERREFS
- GET_INTERREF
- GET_INTERREF_COUNT
- GET_INTERREF_TYPE
- GET_REFERENCED_MODELS
- MOVE_INCOMING_INTERREFS
- MOVE_MODEL_INCOMING_INTERREFS _IGNORE_NONMOVABLE
- QUERY_NON_MOVABLE_MODEL _INCOMING_INTERREFS
- REMOVE_ALL_INTERREFS
- REMOVE_INTERREF
- ***Attribute Profiles***
- CREATE_ATTRPROF_DIRECTORY
- CREATE_ATTRPROF_VERSION
- CREATE_ATTRPROF_VERSION_EXT
- DELETE_ATTRPROF_DIRECTORY
- DELETE_ATTRPROF_THREAD
- DELETE_ATTRPROF_VERSION
- GET_ALL_ATTRPROFS_IN_MODEL
- GET_ALL_ATTRPROF_SUBDIRS
- GET_ALL_ATTRPROF_THREADS_IN_DIR
- GET_ALL_ATTRPROF_VERSIONS_OF_THREAD
- GET_ATTRPROFCLASS_ID
- GET_ATTRPROFCLASS_OF_ATTR
- GET_ATTRPROF_CLASS_OF_THREAD
- GET_ATTRPROF_CLASS_OF_VERSION
- GET_ATTRPROF_DIRECTORY_NAME
- GET_ATTRPROF_SUPERDIR
- GET_ATTRPROF_THREAD_ID_OF_NAME
- GET_ATTRPROF_THREAD_NAME
- GET_ATTRPROF_THREAD_OF_VERSION
- GET_ATTRPROF_VERSIONSTRING
- GET_ATTRPROF_VERSION_USAGE
- GET_REFERENCED_ATTRPROF_VERSION_ID
- GET_ROOT_ATTRPROFDIR_ID
- IS_ATTRPROF_CLASS
- IS_ATTRPROF_THREAD
- IS_ATTRPROF_VERSION
- RENAME_ATTRPROF_DIRECTORY
- RENAME_ATTRPROF_THREAD
- UPDATE_ALL_ATTRPROFS
- UPDATE_SINGLE_ATTRPROF

- ***Expressions***
- EVAL_EXPRESSION
- GET_EXPR_TEXT
- SET_EXPR_TEXT
- GET_EXPR_UPDATE
- SET_EXPR_UPDATE
- UPDATE_ALL_EXPR_ATTRS
- UPDATE_EXPR_ATTRS
- ***Programcall***
- EXECUTE_PROGRAMCALL
- ***Libraries***
- GET_CURRENT_LIBS
- IS_VERSIONING_ENABLED
- DISCARD_LIB
- GET_ALL_APPLIBS
- GET_LIB_ID
- GET_LIB_NAME
- LOAD_LIB
- SAVE_LIBRARY
- ***Model Types***
- GET_ALL_CLASSES_OF_MODE
- GET_ALL_MODELTYPES
- GET_ALL_MODES_OF_MODEL_TYPE
- ***Users***
- GET_USER_PREFERENCES
- SET_USER_PREFERENCES
- ***Error Codes***
- ECODE_TO_ERRTEXT
- ***Copy Buffers***
- CREATE_COPYBUFFER
- DELETE_COPYBUFFER
- FILL_COPYBUFFER
- PASTE_COPYBUFFER
- ***Miscellaneous***
- GET_ENV_STRING
- GET_OS_INFO
- GET_PRODUCT_VERSION
- SET_ENV_STRING

MessagePort "UserMgt"

- ***For both ADOxx toolkits***
- GET_ALL_SYSUSERGROUPS
- GET_ALL_SYSUSER_IDS
- GET_ALL_USERGROUPS
- GET_ALL_USERGROUPS_OF_CURRENT_SYSUSER
- GET_ALL_USERGROUPS_OF_CURRENT_USER
- GET_ALL_USERGROUPS_OF_SYSUSER
- GET_ALL_USERGROUPS_OF_USER
- GET_ALL_USERS
- GET_ALL_USERS_OF_SYSUSERGROUP
- GET_ALL_USERS_OF_USERGROUP
- GET_SYSUSERGROUP_ACCESS_STR
- GET_SYSUSERGROUP_ID
- GET_SYSUSER_ACCESS_STR
- GET_SYSUSER_ID
- GET_SYSUSER_SETTINGS
- GET_USERGROUP_ID
- GET_USER_ACCESS_STR
- GET_USER_ID
- GET_USER_SETTINGS
- USER_SELECT_BOX
- VERIFY_PASSWORD
- ***For both ADOxx toolkits, admin-users only***
- ACTIVATE_READONLY_COMMANDS_FOR_BPMTK
- ADD_SYSUSERS_TO_GROUPS
- ADD_USERS_TO_GROUPS
- CHANGE_SYSUSER_SETTINGS
- CHANGE_USER_SETTINGS
- CREATE_SYSUSER
- CREATE_USER
- REMOVE_SYSUSERS_FROM_GROUPS
- REMOVE_USERS_FROM_GROUP
- SET_SYSUSER_ACCESS_STR
- SET_USER_ACCESS_STR
- ***Commands only for the ADOxx Development Toolkit***
- BALANCE_SYSUSERGROUPS
- CREATE_SYSUSERGROUP
- CREATE_USERGROUP
- DELETE_SYSUSERGROUPS
- DELETE_SYSUSERS
- DELETE_USER
- DELETE_USERGROUPS
- DELETE_USERS
- GET_USERGROUP_ACCESS_STR
- SET_SYSUSERGROUP_ACCESS_STR
- SET_USERGROUP_ACCESS_STR

Figure C.1: Overview of AdoScript MessagePorts and Commands

Appendix D

Documentation

We used Answer Set Programming, the ADOxx Metamodelling Platform and Java for the development of the tool. We used the IntelliJ IDE for writing Java. The versions used for each technology are:

Technology	**Version**
Java	11
IntelliJ	2018.2.5
Clingo	4.5.4
ADOxx	1.5

Table D.1: Development Versions

A PC with the Clingo reasoner and the ADOxx platform installed is needed. The JAR file that implements the translation and connects the 2 end-points will be provided.

In order to use our modelling language, the user has to import the library of our modelling language on the ADOxx Development Toolkit and then assign an ADOxx user instance to that library. Then he can connect to the ADOxx Modelling Toolkit with that user instance's credentials and ECAVI is ready to use.