Institution-based Semantics and Tool Support for the UML

Tobias Rosenberger

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : VERIMAG

Sémantique et outils institutionnels pour le UML

Institution-based Semantics and Tool Support for the UML

Présentée par :

Tobias ROSENBERGER

Direction de thèse :

Saddek BENSALEM PROFESSEUR DES UNIVERSITES - PRATICIEN HOSPITALIER, Université Grenoble Alpes	Directeur de thèse
Markus ROGGENBACH Swansea University	Co-directeur de thèse

Rapporteurs :

Till MOSSAKOWSKI
PROFESSEUR, Otto-von-Guericke-Uniersitat Madeburg
Martin WIRSING
PROFESSEUR, Ludwig-Maximilians-University Munich

Thèse soutenue publiquement le **3 novembre 2022**, devant le jury composé de :

Till MOSSAKOWSKI PROFESSEUR, Otto-von-Guericke-Uniersitat Madeburg	Rapporteur
Martin WIRSING PROFESSEUR, Ludwig-Maximilians-University Munich	Rapporteur
Monika SEISENBERGER ASSOCIATE PROFESSOR, Swansea University	Examinatrice
Marius BOZGA INGENIEUR DE RECHERCHE, CNRS DELEGATION ALPES	Examinateur
Arnold BECKMAN PROFESSEUR, Swansea University	Président

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

preparée dans le cadre d'une cotutelle entre l'Université Grenoble Alpes

et

Swansea University

Spécialité: **Informatique**

Arrêté ministériel : le 6 janvier 2005 -7 août 2006

présentée par

Tobias ROSENBERGER

Thèse dirigée par

Markus Roggenbach

et

Saddek Bensalem

préparée

à Swansea, au sein du **Department of Computer Science** et

à Grenoble, au sein du **Laboratoire VERIMAG**

dans **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique (MSTII)**

«Sémantique et outils institutionnels pour le UML »

Thesis defended

3 November 2022

before a jury consisting of

Arnold Beckmann
Professor at Swansea University
Jury Chair

Monika Seisenberger
Associate Professor at Swansea University
Internal Examiner

Marius Bozga
Research Engineer at Université Grenoble Alpes
Internal Examiner

Till Mossakowski
Professor at Otto-von-Guericke-Universität Magdeburg
External Examiner

Martin Wirsing
Professor emeritus at Ludwig-Maximilians-Universität München
External Examiner

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed T. Rosenberger (candidate)

Date 29/06/2022

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed T. Rosenberger (candidate)

Date 29/06/2022

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed T. Rosenberger (candidate)

Date 29/06/2022

Statement 3

The University's ethical procedures have been followed and, where appropriate, that ethical approval has been granted.

Signed T. Rosenberger (candidate)

Date 29/06/2022

Abstract

Model-based techniques, in particular the Unified Modeling Language (UML), have found wide adoption in the software industry. Formal methods can produce software which is guaranteed to possess desired properties, this being of particular interest where safety or security are concerned. Formal methods have been used very successfully in certain fields like the railway domain (e.g., the verification of the fully automatic lines of the Paris metro system). However, in the wider software industry the adoption of formal methods has been limited. This thesis is part of an effort to ease the uptake of formal methods by enabling a fully formal use of the UML. In pursuing this goal, we want to follow software engineering best practices such as the don't-repeat-yourself principle and modularity.

To achieve this, we use Institution Theory and the Heterogeneous Toolset (HETS). Institution Theory is a principled way to relate different formalisms in a modular way. The Heterogeneous Toolset is based on Institution Theory and implements tool reuse across different formalisms. This is the context into which we wish to integrate the UML, by providing institutional semantics and HETS implementations for the different UML sublanguages. Translations between these UML institutions then give joint semantics. Translations to other established languages implemented in HETS allow us to borrow their tool-support.

The contributions of this thesis to the effort outlined above consist in the development of languages for UML State Machines and Composite Structures and translations to the existing language CASL. We give an account of our languages and translations in terms of Institution Theory, as well as a prototypical implementation in HETS. Further, we verify properties based on specifications in our languages to demonstrate the feasibility of our approach. The experiences from our work on institutionalisation, implementation and verification moreover point to several opportunities for future work.

Abstract

Les techniques basées sur des modèles, en particulier le Unified Modeling Language (UML), ont trouvées une large adoption dans l'industrie du logiciel. Les méthodes formelles peuvent produire des logiciels avec des garanties de certaines propriétés souhaitées, ce qui est particulièrement intéressant concernant des questions de sûreté ou de sécurité. Les méthodes formelles ont été utilisées avec beaucoup de succès dans certains domaines comme la programmation de systèmes ferroviaires (ex. la vérification des lignes automatiques du métro parisien). Malgré ces avantages, l'adoption de méthodes formelles dans la plupart de l'industrie du logiciel a été limitée. Cette thèse fait partie d'une série de travaux visant à faciliter l'adoption de méthodes formelles en permettant une utilisation entièrement formelle d'UML. En poursuivant cet objectif, nous voulons suivre les meilleures pratiques d'ingénierie du logiciel comme le principe de non-répétition et la modularité.

À cette fin, nous utilisons la théorie des institutions et le Heterogeneous Toolset (HETS). La théorie des institutions est une manière systématique et modulaire de mettre en relation de différents formalismes. Le Heterogeneous Toolset est basé sur la théorie des institutions et met en œuvre la réutilisation d'outils à travers différents formalismes. C'est dans ce contexte que nous souhaitons intégrer l'UML, en apportant des sémantiques institutionnels et des implémentations dans HETS pour les différents sous-langages de l'UML. Traductions entre ces institutions donnent alors une sémantique commune. Des traductions vers d'autres langages établis et implémentés dans HETS nous permettent d'emprunter leurs outils.

Les contributions de cette thèse à l'effort décrit ci-dessus consistent dans le développement de langages pour les UML State Machines et les UML Composite Structures et les traductions de ces langages vers le langage CASL. Nous présentons nos langages et traductions en termes de la théorie des institutions, ainsi qu'une implémentation prototypique dans HETS. De plus, nous vérifions les propriétés de quelques spécifications dans nos langages pour une démonstration de la faisabilité de notre approche. Les expériences de nos travaux sur l'institutionnalisation, sur la programmation et la vérification font d'ailleurs ressortir plusieurs possibilités de travaux futurs.

Acknowledgements

I want to take this opportunity to thank some of the many people who have supported and enriched my work and life in this season. First of all, I want to thank my family. Words cannot express my gratitude for what you have been and done for me, not just these last years.

I thank my Swansea supervisor Prof. Markus Roggenbach for inviting me to Wales, for sharing advice, encouragement and perspectives, be it about science, organisation, or life in general. Thank you also for involving me in teaching, for organising social events and for treating all those who work for and with you as humans, not robots. I thank my Grenoble supervisor Prof. Saddek Bensalem for enabling me to spend time in Grenoble, for making me familiar with the ideas of BIP, and for sending me to conferences and meetings with industry partners I thank Prof. Alexander Knapp for working with us. Thank you for all you have taught me about software semantics and verification. Thank you for your good humour in and out of the discussions on our work. I thank my brother Johannes Rosenberger and my friend Marco Eggersmann for proofreading parts of this thesis. I thank Dr. Xiuyi Fan for allowing me to run theorem provers on his server. I thank all the administrative staff who helped me navigate the complications that arise from doing a PhD at two universities and in two countries. I thank Erwin R. Catesbeiana Jr., who joined the research group when he was still wet behind the ears, for all the wisdom he imparted. I thank the examiners for taking the time to read my thesis and to conduct a viva on it, and for their valuable and encouraging feedback.

I am grateful for many interactions with fellow research students and with staff members. It is with a special gratitude that I look back on sharing my office in Grenoble with Dr. Lotfi Mediouni and Dr. Rany Kahil. Thank you for our discussions and for introducing me to everything at the university. Thank you for all the time we spent together outside of work. Without you, I would not have enjoyed the time in Grenoble half as much as I did. May our friendship endure.

I want to thank everyone at Mount Pleasant Baptist Church in Swansea, at Mission Timothée in St. Michel and the Église Chrétienne Evangelique in Grenoble, as well as at the Christian Union in Swansea and the FEU and GBU in Grenoble. I thank you for sharing music and food, excursions and games, for your prayer and encouragement. Thank you for allowing me to see drug addicts find back to an ordered life and generally to see people face life and death in peace. Thank you for helping me see Jesus more clearly in all parts of scripture. Thank you for showing me so many examples of Christian love and service and for making it easy to join in that service. You have given me invaluable help to grow in life and faith.

Again, I thank all of you for walking parts of this journey with me. To speak with the poet: “I don’t know half of you half as well as I should like; and I like less than half of you half as well as you deserve.”

Contents

List of Figures

Chapter 1

Introduction

Contents

Model-based techniques are widely used in industrial software engineering. Chief among the languages used is the Unified Modeling Language (UML) [FL08, TTR+13, WBCW20]. The work presented in this thesis sits at the intersection between model-based software engineering in UML on the one hand and formal specification and verification on the other. It is a contribution to a larger effort to apply software engineering principles, in particular, modularity and the reuse of tools and semantics, to specification and verification, and to make the UML usable in such a context.

Figure 1.1 shows a motivating example system specified by three UML diagrams. The so-called Composite Structure diagram at the top shows how the system is a composite of two

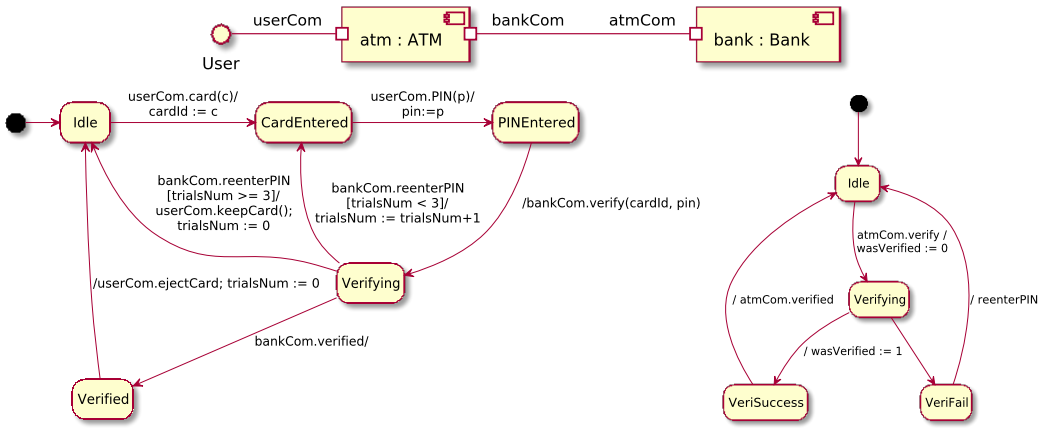


Figure 1.1: UML diagrams for the `ATM` example verified in Sect. 13.2 (implicit completion events omitted): Composite Structure diagram: top; State Machines: left `ATM`, right `Bank`.

subsystems, representing an ATM (Automated Teller Machine) and a bank in a typical usage scenario. The behaviour of the subsystems is then specified by the State Machines at the bottom.

The state machines are for a scenario where a user wants to withdraw money from the ATM, which requires entry of a bank card and pin, as well as their verification by the bank. In such a scenario the bank and its customer (the user) have an obvious interest in ensuring that an ATM will only behave as if a bank card had been verified if the bank has actually performed that verification successfully. This thesis builds towards expressing such properties and systems in a way that is both mathematically rigorous and has tool-support in accordance with the above-mentioned software engineering principles. In Sect. 13.2 we are then able to prove the stated safety property over the example from Fig. 1.1.

To the best of our knowledge, existing work on UML verification falls into two categories:

On the one hand, there are approaches based on model-checking [CE81, QS82], e.g. [SWK+10, tBGM15]. Model-checking is a technique to establish whether a finite-state system satisfies a temporal logic property. Its advantages include full automation and the generation of counterexamples when a property turns out not to hold. However, the applicability of model-checking is limited in theory by the requirement of a finite state space, in practice by the state space explosion problem, where exponential growth of the state space of concurrent systems can quickly exhaust system resources, although there is a large body of work on mitigating this problem [NGYJ21].

On the other hand, there are approaches that verify UML models using interactive theorem provers, e.g. [KFdB+05, Grö10, BBK+04]. These approaches can handle systems with large or even infinite state spaces, but bring with them the cost of a human proof engineer.

We suggest to combine the best aspects of both worlds and apply fully automatic theorem provers to infinite state systems. While acknowledging the well-known limits of possibility and feasibility, we aim to make the fully automatic provers available for such UML verification tasks as they can handle while, at the same time, allowing recourse to interactive techniques where subtasks require them. We want to achieve this without requiring a remodelling of the system for each tool or a reimplementation of each checking or proving technique for UML. To achieve these goals, we work within the verification framework HETS and its theoretical underpinnings, Institution Theory.

The present work is part of a larger effort to provide semantics and tools for the UML in the context of Institution Theory, as first outlined in the position paper [KMR15]. Earlier work as part of this effort includes [KMRG15] (on State Machines) and [KM17] (on Interactions). In [Ros17] we established that the existing approaches to UML State Machines did not satisfy a fundamental condition of Institution Theory, called the Satisfaction Condition. We then proposed that this could best be remedied by creating a multi-modal logic for requiring the presence or absence of certain transitions. A particular State Machine can then be described by sentences requiring that a transition should be possible if the machine mentions it and impossible otherwise[1]. Hennicker et al. [HMK19] propose a similar approach using a simpler logic for event-data systems, as well as an algorithm for systematically computing the sentence

[1] To be precise, the sentences require a semantic transition to be possible if it conforms to some syntactic transition of the machine, and impossible otherwise.

characterising such a system. Their notion of system can be seen as simple UML State Machines without output events, input queues or data parameters. Here, we extend their approach to allow

1. events with data parameters,
2. output events and
3. the combination of State Machines by means of UML Composite Structures, for which purpose we also introduce communication ports and message queues.

We demonstrate the feasibility of our approach by creating a prototypical implementation, integrated into an existing tool landscape, and by carrying out verification tasks on example machines expressed in our formalisms. We will a more extended overview of our approach in Chapter 6, once we have established some background concepts.

1.1 Contributions and Publications

The main contributions of the present work are:

1. the development of two modal logic institutions (EDHML and EDHMLO) for embedding and reasoning about simple UML State Machines (the first lacking output events, the second including them),
2. two embeddings to give semantics to simple UML State Machines in terms of our institutions,
3. the development of an institution for simple UML Composite Structures,
4. the comorphisms from these three institutions to CASL,
5. implementations of our languages and translations in HETS,
6. demonstration of the feasibility of our approach by verifying simple examples which we adapted from the literature and which contain typical challenges.

The most difficult aspects were, on the one hand, the construction of actual institutions, where translation between different vocabularies preserves meaning in the sense required by Institution Theory[2], on the other hand, making the fully automatic provers cope with the inductive reasoning in the presence of many possible axioms from which the prover has to select the relevant ones for each step.

We have published our results in two papers, on which the contribution chapters of this thesis are substantially based: Our work on State Machines without output events, including EDHML and UMLSTATE, was published as [RBKR20]. The extension to State Machines with outputs and our work on Composite Structures has been published as [RKR22]. This includes EDHMLO, UMLSTATEO and UMLCOMP. The results of both papers were obtained in collaboration with our coauthors. Although the delineation of individual contributions in a joint research project is always difficult, we attempt to give an indication thereof at the beginning of each contribution chapter.

[2] Especially the handling of control states is surprisingly difficult.

1.2 Outline of the Remainder of this Thesis

The remainder of this thesis is organised as follows:

In Chapter 2, we give a brief introduction to the UML, including a brief history and the motivations for its creation, a discussion of the model types most relevant for us and an overview of some UML tools. Chapter 3 introduces the notions and notations of Category Theory and Institution Theory which we will be using throughout this thesis. Chapter 4 introduces the specification and proof framework HETS and its central specification language CASL. In Chapter 5 we discuss existing approaches to the verification of models specified in UML.

In Chapter 6, we then give a bird's eye view of our verification approach and its development, i.e., an overview over the contribution chapters of this thesis aimed at readers familiar with the background chapters. Chapter 7 presents the event/data structures underlying our language UMLSTATE, the institution EDHML and a translation from the former to the latter. Chapter 8 introduces a comorphism (a well-behaved translation between institutions) from EDHML to CASL. In Chapter 9, we prepare the connection of State Machines with Composite Structures by extending EDHML to EDHMLO, which adds output events and a notion of ports. We make corresponding extensions to our surface language (now UMLSTATEO) and to its embedding into the logic. Chapter 10 then extends the comorphism, so it now goes from EDHMLO to CASL Chapter 11 introduces our notion of Composite Structures (UMLCOMP) and a comorphism from UMLCOMP to CASL. In Chapter 12, we discuss the implementation of our languages and translations in the context of the Heterogeneous Toolset (HETS). Chapter 13 contains exemplary verification tasks which we carried out on State Machines and Composite Structures. In Chapter 14 we conclude with a summary and some suggestions for future work.

Part I

Background Material

Chapter 2

The Unified Modeling Language

Contents

The Unified Modeling Language (UML) is a modelling methodology and a family of diagram languages which are widely used in software engineering. It was developed in the 1990s to unify the different object-oriented modelling methodologies in use, the number of which had increased to over 50 by 1994. The authors of three of these methods, Grady Booch, Ivar Jacobson and James Rumbaugh, at first adopted ideas from each other's methodologies and then began to actively cooperate in creating a common standard which unified the advantages of each. Their stated goals were as follows [BRJ98]:

1. To model systems, from concept to executable artefact, using object-oriented techniques.

2. To address the issues of scale inherent in complex, mission-critical systems.

3. To create a modelling language usable by both humans and machines.

Since 1997, several versions of the UML have been standardised by the Object Management Group (OMG). The current version of the standard, UML 2.5.1 was released in 2017. UML is widely used in industry and taught in academia. [RHQH+17] report that, although the use of UML in open-source projects has been regarded as relatively uncommon, they were able to extract a dataset of 93000 UML diagrams from over 24000 projects on GitHub.

UML has a common mechanism – the Meta Object Facility (MOF) – to express the abstract grammars of the various model types, as well as the XML based standardised exchange language XMI and, with QVT (Query/View/Transformation), a family of model transformation languages.

Strictly speaking, the abstract grammar defines the notion of a UML model, parts of which can be rendered as diagrams or serialised in XMI. In the following, we will discuss diagrams and model elements without distinction, confident that the reader will remember that details of the graphical presentation are not part of the model as such.

The different UML sublanguages broadly fall into two categories: On the one hand there are static diagrams, which describe, e.g., data models (Class Diagrams) or the hierarchical composition structure of a system (Composite Structure Diagrams). On the other hand, there are dynamic diagrams, which capture run-time behaviour, e.g., State Machines and Interaction Diagrams. Static and dynamic diagrams can be related to one another, two common ways being to specify the behaviour of a component from a Composite Structure Diagram by means of a State Machine, and to specify types in a Class Diagram which are then referred to in a State Machine.

In this thesis, we present formalisations for the basic features of State Machines and Composite Structure Diagrams. UML has an extension mechanism by means of so-called profiles; we do not support this. Adding such support would be straightforward for profiles that only introduce syntactic restrictions. On the other hand, profiles can introduce so-called stereotypes and constraints, the semantics of which can be arbitrarily difficult to model.

In the following, we will introduce a simple security protocol and use it to illustrate our discussion of some UML diagram types.

2.1 Example: A Simple Security Protocol

In this section, we will introduce the example of a simple security protocol adapted from [KCL07], which we will use for the remainder of this chapter to illustrate the different UML diagram types.

This protocol is meant to operate, e.g., at the entrance of a warehouse to allow only authorised goods to enter. Authorisation to enter is expressed by attaching an RFID chip, called a *Tag*, to the goods in question. The *Tag* may confirm that the goods have undergone a customs check, or have been checked to be free of explosives, or any other property we want to ensure for all goods in the warehouse.

Concretely, the protocol can be expressed as follows:

$$\begin{aligned} &1.R \to T: \quad n_R \\ &2.T \to R: \quad ID \oplus n_T,\ h(n_R,k) \oplus n_T \end{aligned}$$

Participants of the protocol are a *Reader* (R) and a *Tag* (T). First, the *Reader* sends the *Tag* a nonce n_R as a challenge. To this message, the *Tag* responds with a pair of values computed by cryptographic operations. The operations used are an exclusive or (written $\oplus$) and a hash function h, both of which can be realised on an RFID chip. The *Tag* holds two variables: ID and k. ID and the key k identify the system. n_T is a second nonce, generated by the *Tag*. The *Reader* authenticates the *Tag* based on the following computation:

$$\begin{aligned} computed\_n_T &= \overbrace{(ID \oplus n_T)}^{\text{received from } Tag} \oplus ID \\ computed\_h &= \underbrace{(h(n_R,k) \oplus n_T)}_{\text{received from } Tag} \oplus computed\_n_T \end{aligned}$$

if $computed\_h = h(n_R,k)$
then $authentication\_success$
else $authentication\_failure$

In the above, the ID, the nonce n_R and the key k are known to the *Reader* beforehand, but the nonce n_T is not. The *Reader* sends its nonce R_n, which need not remain secret, but must not be reused to prevent replay attacks. Based on its knowledge of the preshared secret k, the *Tag* can then construct its reply as indicated above. The *Reader* can then reconstruct first n_T and from this the hash value constructed by the *Tag*, using $\oplus$'s self-inverseness. The computed hash is then compared to the expected one, leading to an authentication success or failure.

"Security protocols address the question of how one communicates 'securely' in an untrusted 'hostile' environment" [RSN22]. By design, the above protocol includes communication between two entities: The *Tag* and the *Reader*. The enviroment is hostile in that anyone might obtain a *Tag*, approach the door from the outside and interpose a device under their control between the *Reader* and the *Tag*. Such an *Intruder* would be a third, undesired protocol participant. In the tradition of Dolev and Yao [DY83], the *Intruder* will be able to read, modify or suppress any messages transmitted between the regular protocol participants.

The above protocol has the authentication property of aliveness, i.e., a protocol run can only be completed if the *Tag* has participated in some way. To be more precise, the *Reader* authenticates a *Tag* if a legitimate *Tag* was involved in some protocol run, although not necessarily the one with the *Reader*. A detailed discussion of these authentication properties can be found in [RSN22].

In the following, we will give an introduction to the UML. Therein we will mostly focus on those UML sublanguages that we will also treat formally later on in this thesis. We take the ability of our chosen sublanguages to encode the protocol example as an indication of its ability to express real-world problems.

2.2 Class Diagrams

Class Diagrams consist of classes and related concepts (see Fig. 2.1 for an example). More precisely, they contain classifiers with their various kinds of relationships. A classifier classifies a set of values called its instances – it is, roughly speaking, UML's concept of a type, although there is also a particular kind of classifiers known as a datatype. The most common kinds of classifier are classes and interfaces, which behave mostly like the classes and interfaces which the reader will know from object-oriented programming.

Figure 2.1 shows some classes and interfaces which model participants in the simple cryptographic protocol from Sect. 2.1. The diagram shows interfaces and a class with inheritance

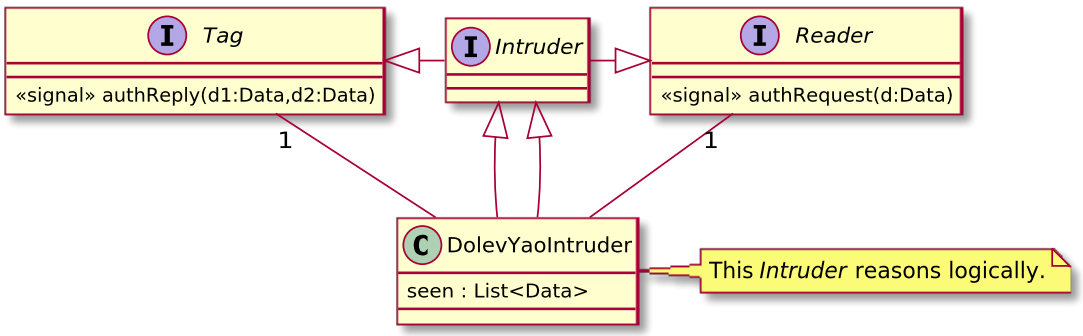


Figure 2.1: A Class Diagram for the security protocol example.

relationships (the arrows) and associations (without arrows). Instances of `Reader` and `Tag` are meant to be the regular participants of the protocol. `Intruder` inherits from both, because it implements the protocols of both – the `Intruder` will act as a man in the middle. A classifier can have features. These include attributes and receptions for signals (as in the example) or for operations[1]. Attributes are just the familiar concept, except that what is an attribute in an implementation may also be expressed by an association in the model. The example shows an attribute `seen` of type `List<Data>`, which contains the data from all messages seen so far by the intruder. The difference between signals and operations is that the former are asynchronous – signals do not return, and execution continues without waiting for them to return – whereas the latter are synchronous – operations can be waited for and can also have a return value.

The example also shows associations between the `DolevYaoIntruder` class and the two interfaces. These associations specify by the numbers at the ends that we expect exactly one `Tag` and exactly one `Reader` to be associated with each `DolevYaoIntruder`. In general, associations can have more than two participants and can make additional requirements, e.g., in which directions the association should be navigable, that instances of one end should be considered aggregates or, as a special case, composites of instances of the other end, etc.

Attached on the right to the `DolevYaoIntruder` class is a comment. Comments can contain arbitrary text and can be attached to (in general: sets of) elements in arbitrary diagrams.

Finally, and importantly for us, a classifier can have an associated Behaviour, i.e., a dynamic diagram (e.g., a State Machine) specifying how it will react to signals and operation calls. However, formalising Class Diagrams themselves is outside the scope of our thesis project[2].

2.3 State Machines

A State Machine is a dynamic diagram, modelling behaviour. That behaviour could, e.g., be associated with a class, specifying the behaviour of its objects. Let us consider as an example Fig. 2.2. At the most basic level, it is a graph: A set of vertices, representing control states, connected by edges, representing state transitions. In addition to the control states, transitions

[1] We only formalise signals, not operations.

[2] There is existing work on institutions for Class Diagrams, e.g., [JKMR13]. Connecting their work to ours could be a direction for future work.

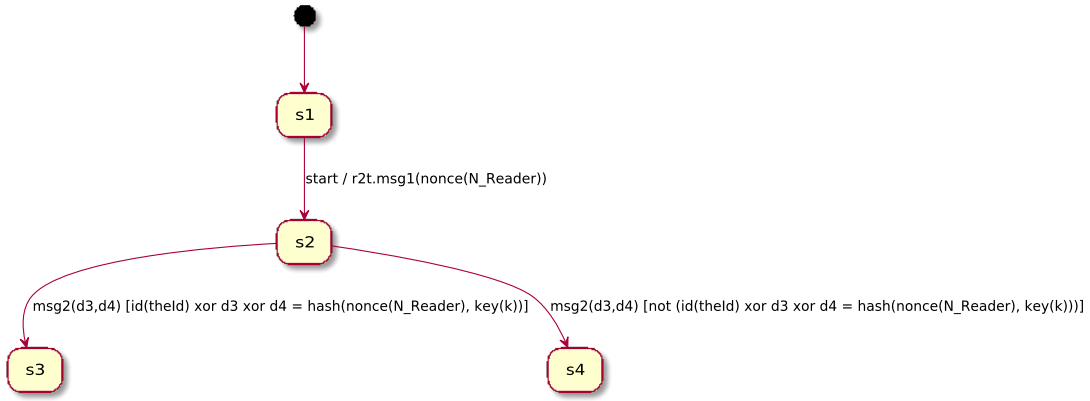


Figure 2.2: State Machine for the *Reader* from the security protocol. The conditions and actions are those from Sect. 2.1, but with an ASCII notation where ID is renamed to *theid* and n_R to *N_Reader*, and the *nonce*s, *key*s and *id*s are explicitly embedded into a common type of message data. Reaching *s3* means a *Tag* has been verified. Reaching *s4* indicates a failed verification.

can depend on and modify a data state. For a State Machine associated with a class, the data state would consist of the attributes of an object of that class. We will refer to the combination of the control and data state as a configuration.

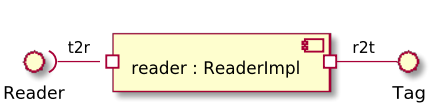
Transitions have labels of the general form $trigger[guard]/action$. Each of the three parts can be omitted. The trigger is an event which can cause the transition to be taken. Explicit triggers will usually be signal receptions or operation calls. The guard is a condition over the data state. Where the right event was received, the truth of the guard, given current data, decides which of the transitions can be taken. An omitted guard is always true. UML does not specify in what language the guards and actions are to be expressed. A mixture of natural language and computer languages is common in UML diagrams, particularly for early design stages, or where diagrams are made only for human consumption. Of course, in this generality, UML cannot be formalised. Our formalisation in Chapter 7 will include a fixed language for expressing guards and actions, which is suitable for tool-supported reasoning.

UML has a variant of State Machines used purely for specifying the input language of a classifier: Protocol State Machines are mostly like ordinary State Machines, but their states can have no hierarchy or other associated behaviour, and their actions are replaced by post-conditions.

In this thesis, we refrain from formalising choice states, hierarchy, state behaviour and Protocol State Machines, as well as the priority rules for completion events. We do not model (synchronous) operation calls, but only (asynchronous) signal receptions.

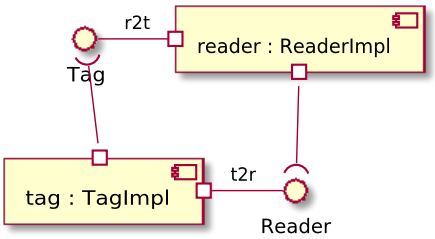
2.4 Composite Structure Diagrams

Composite Structure Diagrams describe so-called structured classifiers. A structured classifier is a classifier – again, by default that means a class – which is considered to have an internal structure. Some examples for our security protocol example can be seen in Fig. 2.3. The internal structure shown by a Composite Structure Diagram consists of a number of roles (drawn as boxes) with ports (little boxes on the borders of the roles) and connectors (shown by lines

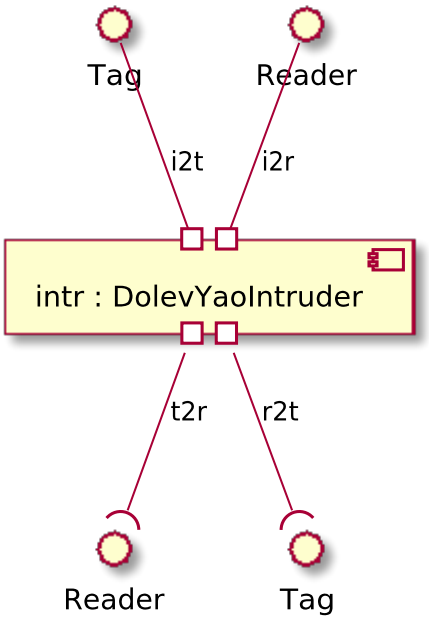


(a) A Reader by itself.

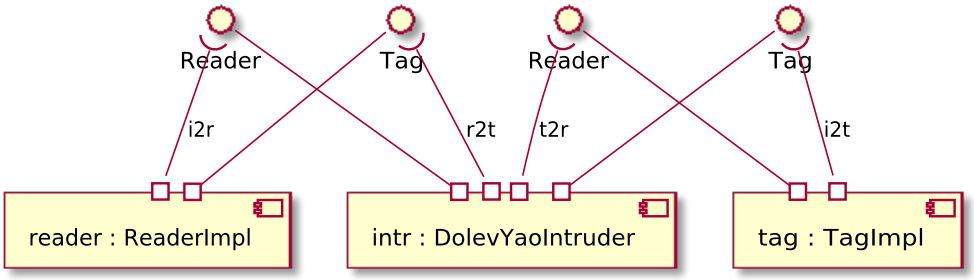
(b) A Tag by itself.



(c) A Reader and Tag put together.



(d) An Intruder by itself.



(e) A Reader, Tag and Intruder put together.

Figure 2.3: Composite Structure Diagrams for our example protocol from Sect. 2.1.

between ports, possibly with a socket/circle pair in the middle). The diagrams can also show interfaces a port offers (the circles) or uses (the sockets).

In the example (Fig. 2.3), we first show systems with just one role; then the intended system, with the roles of *Reader* and *Tag*; and, finally, a system for a man-in-the middle attack, with the *Reader* and the *Tag* communicating through the *Intruder*.

2.5 Tools

In the following we will briefly describe some of the tool landscape available for UML.

```
@startuml
hide empty description
[*] --> s1
s1 --> s2 : start / r2t.msg1(nonce(N_Reader))
s2 --> s3 : msg2(d3,d4) [id(theId) xor d3 xor d4 =
    hash(nonce(N_Reader), key(k))]
s2 --> s4 : msg2(d3,d4) [not (id(theId) xor d3 xor d4 =
    hash(nonce(N_Reader), key(k)))]
@enduml
```

Figure 2.4: PlantUML source for Fig. 2.2

2.5.1 Classical Modelling Tools

As of May 2022, Wikipedia[3] lists no fewer than 49 UML tools, where the features considered relevant for comparison include support for UML2 and the languages for which code generation or reverse engineering exist.

Notably, these features do not include analysis methods of UML2 models, such as simulation, model checking, theorem proving or refinement.

Classical modelling tools provide support for UML models in the strict sense, i.e., following the abstract syntax as prescribed by the MOF-expressed UML metamodel. Apart from editing models, they can check constraints in the Object Constraint Language (OCL) and typically allow some form of model transformations and code generation in certain programming languages. This category includes Enterprise Architect[4], according to [OE20] the most-used UML tool in industry, as well as the Eclipse Modeling Tools[5].

Mostly in the academic realm, some formalisations and formal tools for UML sublanguages exist. More information on these can be found in Chapter 5.

2.5.2 PlantUML

PlantUML [pla] has been used to render all the UML diagrams in this thesis. Also, the formal languages we develop are meant to provide a bridge between the worlds of PlantUML and the verification framework HETS by using a PlantUML-inspired syntax.

PlantUML itself is a tool mostly offering graphical rendering of UML diagrams, based on a simple human-readable textual language (see Fig. 2.4). There are integrations into various other tools, including Eclipse, Microsoft Word, Matlab and WordPress[6]. The author of a diagram can tweak many graphical features, but for the most part, the tool is meant to choose good defaults. For many diagram types, Graphviz is used as the underlying graph layout tool.

[3] https://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools

[4] Proprietary software by Sparx Systems, available at: https://sparxsystems.com/products/ea/index.html

[5] Free software (Eclipse Public License) available from: https://www.eclipse.org/downloads/packages/release/2022-03/r/eclipse-modeling-tools

[6] For a current list of PlantUML integrations, we refer the reader to https://plantuml.com/running

PlantUML is primarily a graphical tool, not a semantic one, and it is easily possible to generate invalid diagrams. There is, however, experimental support for exporting models in XMI.

Chapter 3

Institutions and Comorphisms

Contents

In this chapter, we introduce the concepts of institutions (well-behaved logical formalisms with a notion of vocabulary) and theoroidal comorphisms (well-behaved translations between institutions). But, before we turn to a general discussion, let us first introduce a formalism which we can use as an example.

3.1 Running Example: Modal Logics

Throughout this chapter, we will study a simple modal logic [Lam06, VPMY04] in order to illustrate the concepts of institutions and comorphisms. Modal logic is concerned with the discussion of possible situations. To this end it includes an operator $\langle\rangle\,\phi$ which indicates the possibility of a formula ϕ being true. Such logics can include different modality labels, each of which corresponds to a notion of possibility. For example, in a logic describing the weather, possibility of rain could mean that it rains somewhere or that it rains at some time. Thus, we would have a geographic and a temporal modality label.

For each notion of possibility we immediately gain a corresponding notion of necessity: Whatever cannot possibly be false is necessarily true. This gives us a pair of modalities for each label. In our weather example, geographic necessity of sunshine would mean that the sun shines everywhere, whereas temporal necessity of sunshine would mean that the sun always shines.

Modal logic is a widely applicable method of reasoning for many areas of computer science. These areas include artificial intelligence, database theory, distributed systems, program verification, and cryptography theory.

We will now use a multimodal logic as a running example for building up to the notion of institution.

Let M be a set whose elements we call modality labels and P a nonempty set whose elements we call propositions. Then, sentences are defined by the following grammar, where modality labels are referred to by m and propositions by p:

$$\begin{aligned}\phi, \psi ::= & \; p \\ & \mid \neg\phi \\ & \mid \phi \vee \psi \\ & \mid \langle m \rangle \, \phi\end{aligned}$$

Necessity and the other propositional junctors can then be defined in the obvious way:

$$\begin{aligned}\phi \wedge \psi &:= \neg(\neg\phi \vee \neg\psi) \\ \phi \Rightarrow \psi &:= \psi \vee \neg\phi \\ \phi \Leftrightarrow \psi &:= (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi) \\ [m]\,\phi &:= \neg \langle m \rangle \, \neg\phi\end{aligned}$$

Fixing an arbitrary sentence ϕ, we can moreover define:

$$\begin{aligned}\texttt{true} &:= \phi \vee \neg\phi \\ \texttt{false} &:= \phi \wedge \neg\phi\end{aligned}$$

In order to avoid ambiguity, we use the following operator precedence rules: Negation binds the most tightly; followed by, on one level, conjunction and disjunction; followed by implication; then equivalence; then the modal operators.

For our weather subject, we would use the modality labels $M = \{\texttt{time}, \texttt{place}\}$ and the atomic propositions $P = \{\texttt{sunny}, \texttt{raining}\}$.

Then, we might wish `time` and `place` to relate in such a way that

$$(\langle \texttt{place} \rangle \, [\texttt{time}] \, \texttt{sunny}) \Rightarrow ([\texttt{time}] \, \langle \texttt{place} \rangle \, \texttt{sunny})$$

holds. In words this would be: If there is a place that is always sunny, there is always a sunny place. Of course, the converse would not be tautological.

With some optimism for our food production we might claim:

$$\langle \texttt{place} \rangle \, ((\langle \texttt{time} \rangle \, \texttt{sunny}) \wedge (\langle \texttt{time} \rangle \, \texttt{raining}))$$

In words this would be: There are places where both sunshine and rain do occur.

We will use this notion of modal logics to illustrate the definitions of institutions and theoroidal institution comorphisms. The logics we develop for UML sublanguages are related to the multimodal logics presented here, but somewhat more complicated through their inclusion of constructs from hybrid logic.

3.2 Why use Institutions?

What is a logic? To answer this question formally, Goguen and Burstall developed the concept of an institution [GB83]. Their account contains all the standard ingredients: Signatures, sentences, structures and a notion of satisfaction. However, the collections of structures and, crucially, of signatures, are not simply given as sets, but as categories, i.e., each includes a notion of morphisms. Moreover, there are (several) notions of morphisms between different institutions.

Structure morphisms are, of course, well known in all areas of mathematics. The addition of signature morphisms and morphisms between institutions is particularly interesting for us in a software and systems specification context: By enabling us to reason about changes of notation, they simplify and justify code reuse, refinement and relationships between artefacts in different specification or programming languages. In our general research programme, they allow us to develop small semantics for parts of the UML and combine them by means of a general framework for heterogeneous specification. But already when we are specifying in one particular logic tailored to the problem at hand, we can apply established verification tools developed for different logics, even several different tools within one proof, as long as logics and their translations fulfil the requirements of Institution Theory.

3.3 Categories and Functors

We briefly recapitulate the notions of category and functor and introduce our notations for both.

Definition 3.1 A *category* C is given by

– a class $Obj(\mathrm{C})$, with elements called objects,

– classes $\mathrm{C}(X,Y)$, with elements called morphisms from X to Y, for all $X,Y \in Obj(\mathrm{C})$,

– an identity morphism $1_X : \mathrm{C}(X,X)$ for each $X \in Obj(\mathrm{X})$

– composition[1] operations $\_ \circ_{X,Y,Z} \_ : \mathrm{C}(Y,Z) \times \mathrm{C}(X,Y) \to \mathrm{C}(X,Z)$ for all $X,Y,Z \in Obj(\mathrm{C})$

satisfying

$$1_B \circ_{A,B,B} f = f = f \circ_{A,A,B} 1_A$$
$$(f \circ_{B,C,D} g) \circ_{A,B,D} h = f \circ_{A,C,D} (g \circ_{A,B,C} h)$$

for all $A,B,C \in Obj(\mathrm{C}), f \in \mathrm{C}(C,D), g \in \mathrm{C}(B,C), h \in \mathrm{C}(A,B)$.

Definition 3.2 A *functor* F between two categories C and D (written $F : C \to D$, like for functions) is given by

– an object part $Obj(\mathrm{F}) : Obj(\mathrm{C}) \to Obj(\mathrm{D})$, where we abbreviate $Obj(\mathrm{F})(O)$ to $F(O)$

[1] We will usually write composition omitting indices when types are clear. Where the category is clear, we will write morphisms as $m : X \to Y$.

– morphism parts, which are functions

$$F(O,O') : C(O,O') \to D(F(O),F(O')) \text{ for all } O,O' \in Obj(\mathrm{C}),$$

where we abbreviate $F(O,O')(f)$ to $F(f)$,

preserving identities and composition, i.e.:

$$F(1_O) = 1_{F(O)}$$
$$F(f \circ g) = F(f) \circ F(g)$$

Definition 3.3 A *natural transformation* ν between two functors $F,G : C \to D$ (written $\nu : F \to G$) is given by a morphism $\nu_O : F(O) \to G(O)$ for each object $O \in Obj(\mathrm{C})$, satisfying the conditions that $\nu_Y \circ F(f) = G(f) \circ \nu_X$ for all morphisms $f \in C(X,Y)$.

3.4 Institutions

We are now ready to define institutions, which we will then illustrate by defining an institution for modal logics.

Definition 3.4 An *institution* $\mathcal{I}$ is given by

– a category $\mathrm{Sig}^{\mathcal{I}}$, with objects called signatures,

– a functor $\mathrm{Sen}^{\mathcal{I}}$ assigning

  – to each signature Σ a set $\mathrm{Sen}^{\mathcal{I}}(\Sigma)$ with elements called Σ-sentences and
  – to each signature morphism $\sigma : \Sigma \to \Sigma'$ a function $\mathrm{Sen}^{\mathcal{I}}(\sigma) : \mathrm{Sen}^{\mathcal{I}}(\Sigma) \to \mathrm{Sen}^{\mathcal{I}}(\Sigma')$,

– a functor $\mathrm{Mod}^{\mathcal{I}}$ from signatures to categories[2] called "the reduct functor" assigning

  – to each signature Σ a category with objects called models and
  – to each signature morphism $\sigma : \Sigma \to \Sigma'$ a functor $\mathrm{Mod}^{\mathcal{I}}(\sigma) : \mathrm{Mod}^{\mathcal{I}}(\Sigma') \to \mathrm{Mod}^{\mathcal{I}}(\Sigma)$ going backwards between the categories of models

  and

– a relation $\models^{\mathcal{I}}_{\Sigma} \in \mathrm{Mod}^{\mathcal{I}}(\Sigma) \times \mathrm{Sen}^{\mathcal{I}}(\Sigma)$ for each signature Σ, called the satisfaction relation

fulfilling the satisfaction condition:

$$\mathrm{Mod}^{\mathcal{I}}(\sigma)(\mathcal{M}') \models^{\mathcal{I}}_{\Sigma} \phi \Leftrightarrow \mathcal{M}' \models^{\mathcal{I}}_{\Sigma'} \mathrm{Sen}^{\mathcal{I}}(\sigma)(\phi)$$

for all signatures Σ, Σ' and all Σ' models $\mathcal{M}'$, Σ sentences ϕ and signature morphisms $\sigma : \Sigma \to \Sigma'$.

[2]The reduct functor has the signature $\mathrm{Mod}^{\mathcal{I}} : (\mathrm{Sig}^{\mathcal{I}})^{\mathrm{op}} \to \mathrm{CAT}$. It comes from the opposite category of the category of signatures, i.e., the same category, except that morphisms go in the opposite direction. In other words, the functor $\mathrm{Mod}^{\mathcal{I}}$ reverses the direction of morphisms. For the target of this functor, some standard precautions have to be taken about size issues: It is the category of all (smaller) categories, which has to be large enough to include all categories of models we are interested in. This makes sense in any setting supporting a hierarchy of set universes.

The slogan for the satisfaction condition is: "Truth is invariant under change of notation" [GB83].

We will freely abbreviate $\mathrm{Sen}^{\mathcal{I}}(\sigma)(\phi)$ to $\sigma(\phi)$ and $\mathrm{Mod}^{\mathcal{I}}(\sigma)(\mathcal{M})$ to $\mathcal{M}|\sigma$.

An institution as such does not include a notion of inference. There are established notions of an institution equipped with an inference system, see e.g. [Mos02].

3.5 Example: An Institution for Modal Logics

We will now introduce an institution for the modal logics described in Sect. 3.1.

Definition 3.5 A *modal signature* $\Sigma \in \mathit{Obj}(\mathrm{O})$ is a pair $(P(\Sigma), M(\Sigma))$, where P and M are sets, giving us propositions and modality labels, respectively.

Example 3.6 In our weather example a signature $\Sigma = (P, M)$ could be given as:

$$P = \{\texttt{sunny}, \texttt{raining}\}$$
$$M = \{\texttt{time}, \texttt{place}\}$$

Definition 3.7 A *modal signature morphism* $\sigma : \mathrm{C}(\Sigma, \Sigma')$ is given by

– a proposition part $P(\sigma) : P(\Sigma) \to P(\Sigma')$ and

– a modality part $M(\sigma) : M(\Sigma) \to M(\Sigma')$.

Given signatures $\Sigma, \Sigma', \Sigma''$ and morphisms $\sigma \in \mathrm{C}(\Sigma', \Sigma''), \tau \in \mathrm{C}(\Sigma, \Sigma')$, we define identities and composition:

$$P(1_\Sigma) = 1_{P(\Sigma)} \qquad P(\sigma \circ \tau) = P(\sigma) \circ P(\tau)$$
$$M(1_\Sigma) = 1_{M(\Sigma)} \qquad M(\sigma \circ \tau) = M(\sigma) \circ M(\tau)$$

Example 3.8 Returning again to the weather example, we could have an embedding from a specialised Californian weather logic. There, sunshine, being almost universal, needs no mentioning. Rain, however, when it occurs, is the topic of every conversation. So the Californian weather signature Σ_C would look as follows:

$$P_C = \{\texttt{dripping}\}$$
$$M_C = \{\texttt{when}, \texttt{where}\}$$

This signature can then be embedded into our original weather signature Σ by the following morphism $\sigma = (P(\sigma), M(\sigma)) : \Sigma_C \to \Sigma$:

$$P(\sigma)(\texttt{dripping}) = \texttt{raining}$$
$$M(\sigma)(\texttt{when}) = \texttt{time}$$
$$M(\sigma)(\texttt{where}) = \texttt{place}$$

Definition 3.9 *Modal sentences* are defined by the grammar in Sect. 3.1.

To recall, a sentence would be, e.g.:

$$\langle \texttt{place} \rangle \, ((\langle \texttt{time} \rangle \, \texttt{sunny}) \wedge (\langle \texttt{time} \rangle \, \texttt{raining}))$$

Definition 3.10 Moreover, given any signature $\Sigma \in \mathit{Obj}(\mathrm{O})$, we define a category $\mathrm{Mod}(\Sigma)$ of all *structures* over Σ: An object $\mathcal{M} \in \mathit{Obj}(\mathrm{Mod}(\Sigma))$ is given by a Kripke frame, i.e.,

– a set $W(\mathcal{M})$, with elements called worlds,

– relations $R(\mathcal{M})(m) \subseteq W(\mathcal{M}) \times W(\mathcal{M})$ called (world) accessibility relations for all modality labels $m \in M(\Sigma)$ and

– a relation $\mathit{Sat}(\mathcal{M}) \subseteq W(\mathcal{M}) \times P(\Sigma)$ called proposition satisfaction.

Between two structures, there is a single morphism $\leq! : \mathcal{M} \to \mathcal{M}'$ if the world set, accessibility relations and proposition satisfaction in $\mathcal{M}$ are all subsets of their counterparts in $\mathcal{M}'$. Otherwise, the set of morphisms is empty.

Example 3.11 Over our weather signature, worlds could be given as the Cartesian product of times and places, i.e. $W(\mathcal{M}) = W_t \times W_p$. Figure 3.1 shows a possible set of times with its accessibility relation; Fig. 3.2 shows the same for places. In these cases the relations would be extended to worlds by ignoring the other tuple component (places or times).

The examples given are very simple in at least two ways that cannot be generalised to arbitrary multimodal logics:

1. Each world is accessible from each. To justify a more interesting relation, we could consider another modality `nearby` which holds in all nearby cities, defined as those reachable from our current location by a train journey. We could then indicate sunshine somewhere around here ($\langle \texttt{nearby} \rangle \, \texttt{sunny}$) or all around here ($[\texttt{nearby}] \, \texttt{sunny}$). Figure 3.3 gives an example relation for this modality. Likewise, in the case of time, modalities with more restrictive accessibility can be imagined, and are commonly used in temporal logics. In theories over our old weather signature, we might wish to use axioms ensuring this all-connectedness.

2. Time and place are effectively independent. If we wanted a relativistic weather logic, we would need to consider the space-time continuum. Worlds could not then be given by a Cartesian product of independent places and times. The two accessibility relations would then need to be stated directly over whole set of worlds, and time and space could interact. Such interactions between accessibility relations are common in practice. However, where the relations are independent, human and automated reasoning can often be simplified by exploiting this natural modularity.

The reader can easily imagine a proposition satisfaction relation for the worlds and the weather propositions given.

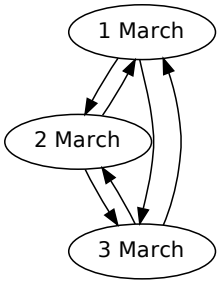


Figure 3.1: A possible accessibility relation for time over our weather signature.

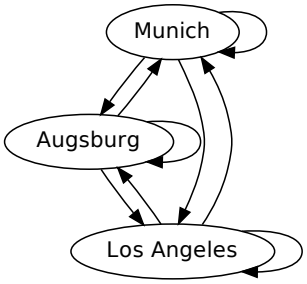


Figure 3.2: A possible accessibility relation for places over our weather signature.

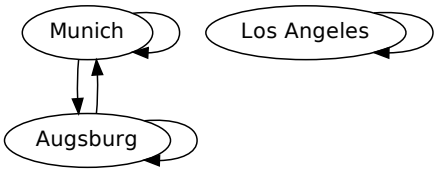


Figure 3.3: A possible accessibility relation $R(\mathcal{M})(\texttt{nearby})$ over an extended weather signature.

Definition 3.12 We now define *translation* for multimodal sentences as the structural closure of proposition and modality translation:

$$\begin{aligned}\sigma(p) &= P(\sigma)(p)\\ \sigma(\neg\phi) &= \neg\sigma(\phi)\\ \sigma(\phi \vee \psi) &= \sigma(\phi) \vee \sigma(\psi)\\ \sigma(\langle m\rangle\,\phi) &= \langle M(\sigma)(m)\rangle\,\sigma(\phi)\end{aligned}$$

Example 3.13 The Californian weather sentence

$$\langle\texttt{when}\rangle\,\texttt{dripping}$$

translates into

$$\langle\texttt{time}\rangle\,\texttt{raining}$$

.

Definition 3.14 *Reduction* for multimodal structures preserves the world set and transports accessibility relations and proposition satisfaction.

$$W(\mathcal{M}'|\sigma) = W(\mathcal{M}')$$

$$R(\mathcal{M}'|\sigma)(m)(v,w) \iff R(\mathcal{M}')(M(\sigma)(m))(v,w)$$
$$Sat(\mathcal{M}'|\sigma)(w,p) \iff Sat(\mathcal{M}')(w,P(\sigma)(p))$$

Example 3.15 Reducing general weather models to Californian weather models is slightly more involved than embedding sentences. A model given by

$$\begin{aligned} W(\mathcal{M}') &= \{(\text{Frisco}, \text{April } 1)\} \\ R(\mathcal{M}')(\texttt{time}) &= \{((\text{Frisco}, \text{April } 1), (\text{Frisco}, \text{April } 1))\} \\ R(\mathcal{M}')(\texttt{place}) &= \{((\text{Frisco}, \text{April } 1), (\text{Frisco}, \text{April } 1))\} \\ Sat(\mathcal{M}') &= \{((\text{Frisco}, \text{April } 1), \texttt{raining}), \\ &\qquad ((\text{Frisco}, \text{April } 1), \texttt{sunny})\} \end{aligned}$$

reduces to

$$\begin{aligned} W(\mathcal{M}') &= \{(\text{Frisco}, \text{April } 1)\} \\ R(\mathcal{M}')(\texttt{when}) &= \{((\text{Frisco}, \text{April } 1), (\text{Frisco}, \text{April } 1))\} \\ R(\mathcal{M}')(\texttt{where}) &= \{((\text{Frisco}, \text{April } 1), (\text{Frisco}, \text{April } 1))\} \\ Sat(\mathcal{M}') &= \{((\text{Frisco}, \text{April } 1), \texttt{dripping})\}. \end{aligned}$$

So we have dropped one proposition and "renamed" the accessibility relations as well as the remaining proposition. If the Californian logic additionally had no concept of time, we would drop one accessibility relation, but worlds would still carry time information, and the remaining accessibility relation would still only be renamed without being modified.

In any of these variations, consider a sentence the translation of which holds in all models of the translated signature. Then, by the satisfaction condition, the sentence must already hold in all those models over the original signature which arise as reducts of some model over the translated signature. In fact, all Californian weather models arise this way, so a sound proof system for general weather sentences would give rise to another sound proof system for Californian weather sentences. A general theory of such borrowings can be found in [CM97]. For our proof case studies, we establish a borrowing principle in Prop. 10.3.

Definition 3.16 The *satisfaction relation* for multimodal sentences and structures is defined by:

$$\begin{aligned} \mathcal{M} \models \phi &\iff \text{for all } w \in W(\mathcal{M}), \text{ we have } (M,w) \models \phi \\ (\mathcal{M},w) \models p &\iff Sat(\mathcal{M})(w,p) \\ (\mathcal{M},w) \models \neg\phi &\iff (\mathcal{M},w) \not\models \phi \\ (\mathcal{M},w) \models \phi \vee \psi &\iff (\mathcal{M},w) \models \phi \text{ or } (\mathcal{M},w) \models \psi \\ (\mathcal{M},w) \models \langle m \rangle \phi &\iff \text{exists } v \in W(\mathcal{M}) \\ &\qquad \text{s.t. } R(\mathcal{M})(m)(w,v) \\ &\qquad \text{and } (\mathcal{M},v) \models \phi \end{aligned}$$

Then the following theorem establishes the satisfaction condition by a typical proof using structural induction over the grammar of sentences:

Theorem 3.17 Let $\Sigma,\Sigma' \in \mathrm{Sig}(C)$, $\sigma \in C(\Sigma,\Sigma')$, $\mathcal{M}' \in \mathrm{Mod}(\Sigma')$, $w \in W(\mathcal{M}'|\sigma) \subseteq W(\mathcal{M}')$. Then the following equivalence holds:

$$(\mathcal{M}',w) \models \sigma(\phi) \iff (\mathcal{M}'|\sigma,w) \models \phi$$

Proof. We proceed by induction over ϕ:
case p:

$$\begin{aligned}
& (\mathcal{M}',w) \models \sigma(p) \\
\iff & \text{[def. form. trans]} \\
& (\mathcal{M}',w) \models P(\sigma)(p) \\
\iff & \text{[def. sat. rel.]} \\
& Sat(\mathcal{M}')(w,P(\sigma)(p)) \\
\iff & \text{[def. reduct]} \\
& Sat(\mathcal{M}'|\sigma)(w,p) \\
\iff & \text{[def. sat. rel.]} \\
& (\mathcal{M}'|\sigma,w) \models p
\end{aligned}$$

case $\neg\phi$:

$$\begin{aligned}
& (\mathcal{M}',w) \models \sigma(\neg\phi) \\
\iff & \text{[def. form. trans]} \\
& (\mathcal{M}',w) \models \neg(\sigma(\phi)) \\
\iff & \text{[def. sat. rel.]} \\
& ((\mathcal{M}',w) \not\models \neg(\sigma(\phi))) \\
\iff & \text{[IH]} \\
& ((\mathcal{M}'|\sigma,w) \not\models \phi) \\
\iff & \text{[def. sat. rel.]} \\
& (\mathcal{M}'|\sigma,w) \models \neg\phi
\end{aligned}$$

case $\phi \vee \psi$:

$$\begin{aligned}
& (\mathcal{M}',w) \models \sigma(\phi \vee \psi) \\
\iff & \text{[def. form. trans]} \\
& (\mathcal{M}',w) \models (\sigma(\phi)) \vee (\sigma(psi)) \\
\iff & \text{[def. sat. rel.]} \\
& (\mathcal{M}',w) \models (\sigma(\phi))or(\mathcal{M}',w) \models (\sigma(\psi)) \\
\iff & \text{[IH]} \\
& ((\mathcal{M}'|\sigma,w) \models \phi)or((\mathcal{M}'|\sigma,w) \models \psi) \\
\iff & \text{[def. sat. rel.]}
\end{aligned}$$

$$(\mathcal{M}'|\sigma, w) \models \phi \vee \psi$$

case $\langle m \rangle \phi$:

$$\begin{array}{ll}
& (\mathcal{M}', w) \models \sigma(\langle m \rangle \phi) \\
\iff & [\text{def. form. trans.}] \\
& (\mathcal{M}', w) \models \langle M(\sigma)(m) \rangle \sigma(\phi) \\
\iff & [\text{def. sat. rel.}] \\
& (\text{exists } v' \in W(\mathcal{M}') \\
& \quad \text{s.t. } R(\mathcal{M}')(M(\sigma)(m))(w, v') \\
& \quad \text{and } (\mathcal{M}', v') \models \sigma(\phi))) \\
\iff & [\text{ from def. form. trans.:} \\
& \quad R(\mathcal{M}')(M(\sigma)(m))(w, v') \Rightarrow v' \in W(\mathcal{M}'|\sigma) \,] \\
& (\text{exists } v \in W(\mathcal{M}'|\sigma) \\
& \quad \text{s.t. } R(\mathcal{M}')(M(\sigma)(m))(w, v) \\
& \quad \text{and } (\mathcal{M}', v) \models \sigma(\phi)) \\
\iff & [\text{ IH }] \\
& (\text{exists } v \in W(\mathcal{M}'|\sigma) \\
& \quad \text{s.t. } R(\mathcal{M}')(M(\sigma)(m))(w, v) \\
& \quad \text{and } ((\mathcal{M}'|\sigma, v) \models \phi)) \\
\iff & [\text{def. reduct }] \\
& (\text{exists } v \in W(\mathcal{M}'|\sigma) \\
& \quad \text{s.t. } R(\mathcal{M}'|\sigma)(m)(w, v) \text{ and } ((\mathcal{M}'|\sigma, v) \models \phi)) \\
\iff & [\text{def. sat. rel.}] \\
& (\mathcal{M}'|\sigma, w) \models \langle m \rangle \phi
\end{array}$$

□

3.6 Comorphisms

Definition 3.18 An (institution) *comorphism* [GR02] $\nu : \mathcal{I} \to \mathcal{J}$ between institutions $\mathcal{I}$ and $\mathcal{J}$ is given by

– a functor $\nu^{\mathrm{Sig}} : \mathrm{Sig}^{\mathcal{I}} \to \mathrm{Sig}^{\mathcal{J}}$ translating signatures,

– a natural transformation $\nu^{\mathrm{Sen}} : \mathrm{Sen}^{\mathcal{I}} \to (\mathrm{Sen}^{\mathcal{J}} \circ \nu^{\mathrm{Sig}})$ translating sentences and

– a natural transformation $\nu^{\mathrm{Mod}} : (\mathrm{Mod}^{\mathcal{J}} \circ (\nu^{\mathrm{Sig}})^{\mathrm{op}}) \to \mathrm{Mod}^{\mathcal{I}}$ reducing models

fulfilling the satisfaction condition $\mathcal{M}' \models \mathrm{Sen}(\nu)(\phi) \iff \mathcal{M}'|\nu \models \phi$.

Note that as before, sentences are translated forward and structures reduced backward. Signatures and their morphisms are translated forward.

We will actually use a slight variation of the notion of comorphism, which allows properties that are inherent in a signature of the source institution to be expressed by sentences in the target institution. To make this precise, we first extend signatures to theory presentations:

Definition 3.19 A *theory presentation* $T = (\Sigma, \Phi)$ in the institution $\mathcal{I}$ consists of a signature $\Sigma \in |\mathrm{Sig}^{\mathcal{I}}|$, also denoted by $Sig(T)$, and a set of sentences $\Phi \subseteq \mathrm{Sen}^{\mathcal{I}}(\Sigma)$. Its *model class* $\mathrm{Mod}^{\mathcal{I}}(T)$ is the class $\{M \in \mathrm{Mod}^{\mathcal{I}}(\Sigma) \mid M \models^{\mathcal{I}}_{\Sigma} \phi \text{ for all } \phi \in \Phi\}$ of the Σ-structures satisfying the sentences in Φ. A *theory presentation morphism* $\sigma : (\Sigma, \Phi) \to (\Sigma', \Phi')$ is given by a signature morphism $\sigma : \Sigma \to \Sigma'$ such that $M' \models^{\mathcal{I}}_{\Sigma'} \mathrm{Sen}^{\mathcal{I}}(\sigma)(\phi)$ for all $\phi \in \Phi$ and $M' \in \mathrm{Mod}^{\mathcal{I}}(\Sigma', \Phi')$. Theory presentations in $\mathcal{I}$ and their morphisms form the category $\mathrm{Pres}^{\mathcal{I}}$.

Note that Φ can be an infinite set of sentences, making theory presentations more general than finitary notions of a specification.

Definition 3.20 A *theoroidal institution comorphism*[3] $\nu = (\nu^{\mathrm{Sig}}, \nu^{\mathrm{Mod}}, \nu^{\mathrm{Sen}}) : \mathcal{I} \to \mathcal{J}$ is given by

- a functor $\nu^{\mathrm{Pres}} : \mathrm{Sig}^{\mathcal{I}} \to \mathrm{Pres}^{\mathcal{J}}$ inducing the functor $\nu^{\mathrm{Sig}} = \mathrm{Sig} \circ \nu^{\mathrm{Sig}} : \mathrm{Sig}^{\mathcal{I}} \to \mathrm{Sig}^{\mathcal{J}}$ on signatures,
- a natural transformation $\nu^{\mathrm{Mod}} : \mathrm{Mod}^{\mathcal{J}} \circ (\nu^{Sig})^{\mathrm{op}} \to \mathrm{Mod}^{\mathcal{I}}$ on models and structures, and
- a natural transformation $\nu^{\mathrm{Sen}} : \mathrm{Sen}^{\mathcal{I}} \to \mathrm{Sen}^{\mathcal{J}} \circ \nu^{\mathrm{Sig}}$ on sentences,

such that for all $\Sigma \in |\mathrm{Sig}^{\mathcal{I}}|$, $M' \in |\mathrm{Mod}^{\mathcal{J}}(\nu^{\mathrm{Sig}}(\Sigma))|$, and $\phi \in \mathrm{Sen}^{\mathcal{I}}(\Sigma)$ the following *satisfaction condition* holds:

$$\nu^{\mathrm{Mod}}_{\Sigma}(M') \models^{\mathcal{I}}_{\Sigma} \phi \iff M' \models^{\mathcal{J}}_{\nu^{\mathrm{Sig}}(\Sigma)} \nu^{\mathrm{Sen}}(\Sigma)(\phi) \,.$$

A theory presentation (Σ, Φ) over the institution $\mathcal{I}$ is *translated* via a theoroidal institution comorphism $\nu : \mathcal{I} \to \mathcal{J}$ into the theory presentation $\nu^{\mathrm{Pres}}(\Sigma, \Phi) = (\Sigma_\nu, \Phi_\nu \cup \nu^{\mathrm{Sen}}_{\Sigma}(\Phi))$ over $\mathcal{J}$ where $\nu^{\mathrm{Sig}}(\Sigma) = (\Sigma_\nu, \Phi_\nu)$ and $\nu^{\mathrm{Sen}}_{\Sigma}(\Phi) = \{\nu^{\mathrm{Sen}}_{\Sigma}(\phi) \mid \phi \in \Phi\}$.

3.6.1 Example: A Theoroidal Comorphism for the Standard Translation

As an example of a theoroidal comorphism, we present the *standard translation* from our institution for multimodal logics into an institution for first-order logic. We will not present the latter institution in detail. The interested reader can find an example of a first-order institution in [DM16].

The standard translation is the established way of relating modal logics to predicate logic [BS95]. We will use an adapted version of it in our treatment of State Machines. Intuitively, it

[3] What we call theoroidal institution comorphism was introduced as simple map of logics [Mes89]. A similar notion using theories (closed under semantic consequence) instead of presentations (arbitrary sets of sentences) is called simple theroidal institution comorphism in [GR02], where there is also a notion of theoroidal comorphism mapping from, as well as to, theories.

is best understood by observing that the sentence translation directly mirrors the semantics of multimodal logic.

Translation of signatures. Assume a multimodal signature $\Sigma = (P, M)$.

We will here take a first-order signature $\Sigma^{FOL} = (S, F, Pr)$ to consist of three sets, giving us sorts (S), function symbols (F) and predicate symbols (Pr), respectively, which are compatible in the obvious ways. In our translation of modal logic signatures, we have just one sort, namely that of worlds, no function symbols, per proposition one predicate symbol $Sat(p)$ for proposition satisfaction, and per modality label m one accessibility relation $R(m)$. Observe how this mirrors the definition of multimodal models.

$$\begin{aligned}
\nu^{Sig}(\Sigma) &= ((S, F, Pr), \Phi) \\
S &= \{W\} \\
F &= \{\} \\
Pr &= \{Sat(p) : W \mid p \in P\} \cup \{R(m) : W \times W \mid m \in M\} \\
\Phi &= \{\}
\end{aligned}$$

If we wished to have axioms restricting the class of legal accessibility relations, as mentioned above, these could go into Φ. Here, however, we are going to allow all relations of appropriate type.

Translation of sentences. For sentences, we first give a version of the translation indexed over a variable for the current world. This version we define by recursion over the argument sentence. Then, as needed for the comorphism, we define a version without the index by universally quantifying over worlds.

$$\begin{aligned}
\nu^{\mathrm{Sen}}_w(p) &= Sat(p)(w) \\
\nu^{\mathrm{Sen}}_w(\neg\phi) &= \neg\nu^{\mathrm{Sen}}_w(\phi) \\
\nu^{\mathrm{Sen}}_w(\phi \vee \psi) &= \nu^{\mathrm{Sen}}_w(\phi) \vee \nu^{\mathrm{Sen}}_w(\psi) \\
\nu^{\mathrm{Sen}}_w(\langle m \rangle \phi) &= \exists v : W.R(m)(w, v) \wedge \nu^{\mathrm{Sen}}_v(\phi), \\
&\qquad \text{where } v \text{ is a variable not free in } \phi \\
\nu^{\mathrm{Sen}}(\phi) &= \forall w : W.\nu^{\mathrm{Sen}}_w(\phi)
\end{aligned}$$

Example 3.21

$$(\langle \texttt{place} \rangle [\texttt{time}] \texttt{raining}) \Rightarrow ([\texttt{time}] \langle \texttt{place} \rangle \texttt{raining})$$

would translate to

$$\begin{aligned}
&\forall w_1 : W.(\exists w_2 : W.R(\texttt{place})(w_1, w_2) \wedge \\
&\qquad\qquad \neg\exists w_3 : W.R(\texttt{time})(w_2, w_3) \wedge \neg(Sat(\texttt{raining})(w_3))) \\
&\quad \Rightarrow (\neg\exists w_2 : W.R(\texttt{place})(w_1, w_2) \wedge \neg
\end{aligned}$$

$$\exists w_3 : W.R(\texttt{time})(w_2, w_3) \wedge Sat(\texttt{raining})(u)).$$

We can observe that the sentences of our specialised logic are more succinct than the corresponding first-order sentences. Note, however, that the latter can be somewhat shortened by adding a specialised translation for the necessity modalities.

Reduction of structures. First-order structures are functions that take each symbol to its interpretation: Sorts to nonempty sets, and function and predicate symbols to appropriate subsets of Cartesian products of interpreted sorts. We only have to handle structures that conform to the translation of the modal signature (P, M), which makes it straightforward to recover our multimodal structures:

$$\nu^{\mathrm{Mod}}(\mathcal{M}') = (\mathcal{M}'(W), \{\mathcal{M}'(R(m)) \mid m \in M\}, \{\mathcal{M}'(Sat(p)) \mid p \in P\})$$

Chapter 4

The Algebraic Specification Language CASL and its Toolset HETS

Contents

This chapter introduces the reader to the technical setting for our proof examples and our implementation: first to the Common Algebraic Specification Language (CASL); then to the Heterogeneous Toolset (HETS), a specification toolset based on Institution Theory in which CASL is the central language; then to Haskell, the language in which HETS, including our extensions, are implemented; and finally to Parsec, the Haskell library used for all parsing needs in HETS.

4.1 The CASL Language Family

In 1995, the Common Framework Initiative (CoFI) [Mos97] was started to reduce the fragmentation of the field of algebraic specification. Convinced that there had been enough experimentation for the community to know which features, convictions and tastes should be considered in such an effort, CoFI set out to build a core language that should be acceptable to the whole community with comprehensive documentation and agreed principles for defining restrictions and extension of that language. In 1997 a language design was proposed, featuring abstract syntax and semantics, but deliberately leaving the concrete syntax open. Version 1.0 of the Common Algebraic Specification Language (CASL) was adopted in 2001. The current version (1.0.2) was adopted in 2003, with no further changes planned [BM03].

The language CASL is the core output of CoFI. The comprehensive documentation, which the earlier, fragmented efforts were often missing, comes in the form of two manuals: The *User Manual* [BM03] is an example-based introduction to the language and to some supporting tools.

It aims to be didactical rather than complete, but is still more comprehensive than we can be in this thesis. The *Reference Manual*, on the other hand, aims for completeness above all else. It provides a language overview, formal grammars for the abstract and concrete syntax of CASL, as well as the formal semantics [RMS03].

Several aspects of CASL can be understood and used largely independently from one another: *Basic Specifications* are a notation for many-sorted first-order theories with sort generation constraints, subsorting, partiality and equality. Much effort went into combining the availability of total and partial functions on the one hand with subsorting on the other hand. The language of *Structured Specifications* provides constructs for parameterising, modifying (by hiding and renaming of symbols) and combining Basic Specifications. The structuring here applies only to the presentation of the specification and does not affect the models. In contrast to this, *Architectural Specifications* can require structure and parameterisation of models (pragmatically speaking: of conforming software artefacts). *Libraries* are sequences of named specifications, where earlier specifications can be included by name in later Structured and Architectural Specifications.

Consider the specification of natural numbers in Fig. 4.1. Everything between `Nat =` and **then** is a Basic Specification. A Basic Specification introduces sorts, operation symbols and predicate symbols with type signatures, and axioms. In our natural numbers example, this is the order in which they are given, with one exception: With the free type construction we simultaneously introduce operation symbols for the constructors and axioms that requires the sort to be freely generated by the constructors. A sort could also be introduced with just generatedness or with no generation constraints at all. The meaning of these constraints is as usual, but can be summarised by the slogan "no junk, no confusion": essentially, for a generated type, all values can be written as constructor terms ("no junk"). This establishes an induction principle, which actually takes us somewhat beyond first-order logic. For free types, different constructor terms additionally have to stand for different values ("no confusion"). For the precise semantics, we refer the reader to the *Reference Manual* [RMS03].

Turning again to the example, we see three sections separated by **then**. This is the first structuring concept we will consider, that of extension. Each section here is spelled out directly, but we could also refer to other specifications by name. The later sections add symbols and axioms that can refer back to the earlier sections, e.g., a new operation can have a sort introduced in an earlier section. The reader will further notice that that axioms can be given names (here: `decimal_def`) and that **then** can be followed by annotations about the relationship of the following to the previous section. Here, the second section is a definitional extension of the first, and third merely adds implied axioms – when we come to proving, these will be our proof obligations.

Besides extension, CASL offers the structuring concepts of union, renaming and hiding of symbols, naming of specifications, and generic (parameterised) specifications, which will not here be discussed in detail.

```
%prec({__+__}<{__*__})%
%left_assoc(__+__, __*__)%
%number __@@__

spec Nat =
  free type Nat ::= 0 | suc (Nat)
  op __ + __ : Nat * Nat -> Nat;
     __ * __ : Nat * Nat -> Nat;
  pred __<=__, __ < __, __>__, __>=__: Nat * Nat

forall n,m,o: Nat
. 0 + n = n
. suc(n) + m = suc(n+m)

. 0 * n = 0
. suc(n) * m = m + (n * m)

. 0 <= n
. not suc(m) <= 0
. suc(m) <=  suc(n) <=> m <= n

. m >=n <=> n <= m
. m < n <=> m <= n /\ not m=n
. m > n <=> n < m

then %def
%% Operations to represent natural numbers with digits:
ops 1: Nat = suc(0);
    2: Nat = suc(1);
    3: Nat = suc(2);
    4: Nat = suc(3);
    5: Nat = suc(4);
    6: Nat = suc(5);
    7: Nat = suc(6);
    8: Nat = suc(7);
    9: Nat = suc(8);
    __@@__(m: Nat; n: Nat): Nat = m * suc(9) + n %(decimal_def)%

then %implies
op __ + __ : Nat * Nat -> Nat, assoc, comm, unit 0;
   __ * __ : Nat * Nat -> Nat, assoc, comm, unit 1;
end
```

Figure 4.1: A CASL specification for natural numbers.

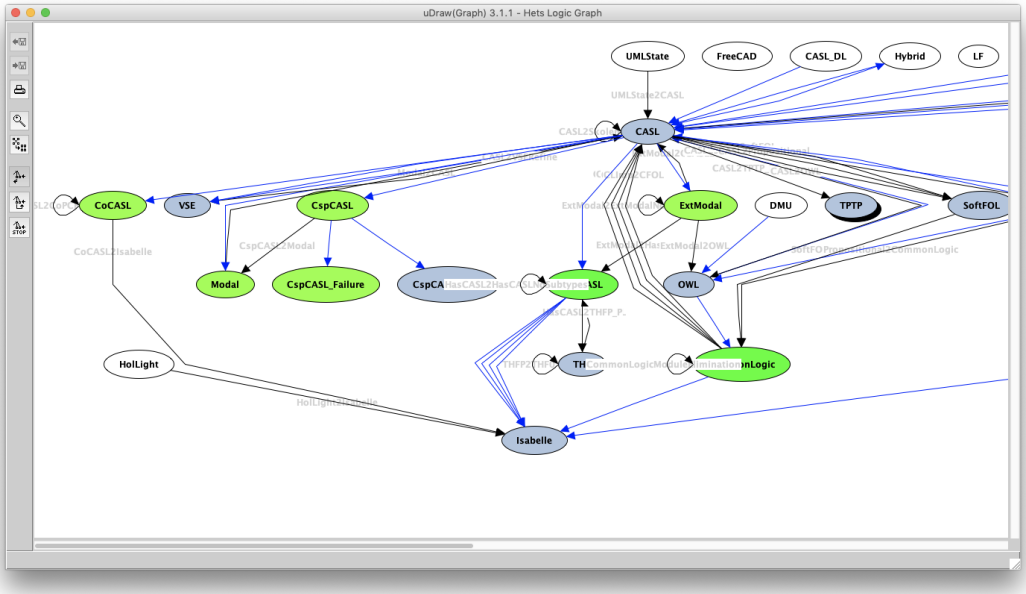


Figure 4.2: The HETS language graph, showing UMLSTATE with the translation to CASL . Green as opposed to white indicates higher stability of the logic; blue indicates logics with direct prover support[MMCL14].

4.2 Heterogeneous Tooling with HETS

The Heterogeneous Toolset HETS[1,2] [MMCL14, MML07a] is a verification suite built around CASL and Institution Theory. It is written in the functional programming language Haskell [M+10], and contains parsers for various formal languages, as well as translations between those languages. HETS includes bindings to many existing verification tools. It provides proof support for heterogeneous specifications, i.e., specifications using more than one language. This heterogeneous verification is meaningful as long as the languages involved are institutions and the translations are theoroidal comorphisms, as introduced in Chapter 3.

We will now briefly discuss some screenshots and graphics to illustrate some principles of HETS and our use of it, simultaneously giving the reader an impression of its user interface.

Figure 4.2 shows HETS with a visualisation of its language graph. UMLSTATE (leftmost in the top row) is a language we implement in this thesis for UML State Machines – more on this in Chapter 12. It is easy to see the central place of CASL as both the source and target of many translations. Our approach fits neatly into that picture: We use translations via CASL to borrow [CM97] existing verification tools for our new languages.

Specifications and verification tasks are represented as *development graphs*. An example of such a graph is shown in Fig. 4.3. A development graph represents a library, which is mostly like a CASL library, but allows the specifications contained in the library to be written in any of the logics implemented in HETS, not just CASL itself. Here we see in action the independence of the

[1] `https://hets.eu/`

[2] In active development at `https://github.com/spechub/Hets`.

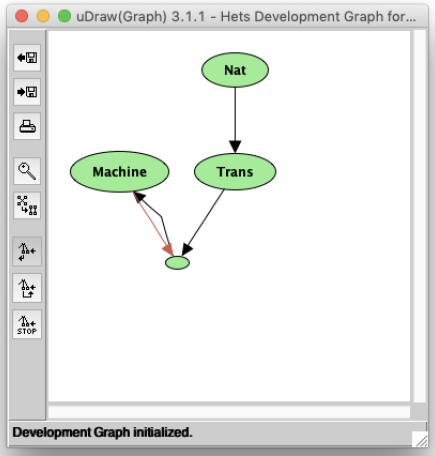


Figure 4.3: A development graph in HETS.

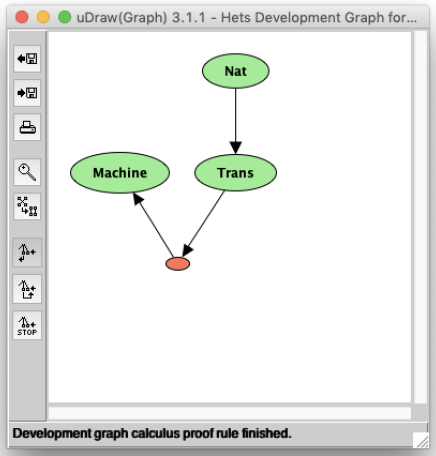


Figure 4.4: The development graph from Fig. 4.3 after applying the *global decomposition* proof rule.

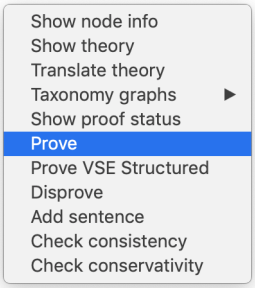


Figure 4.5: A context menu of actions possible on the red node from Fig. 4.4.

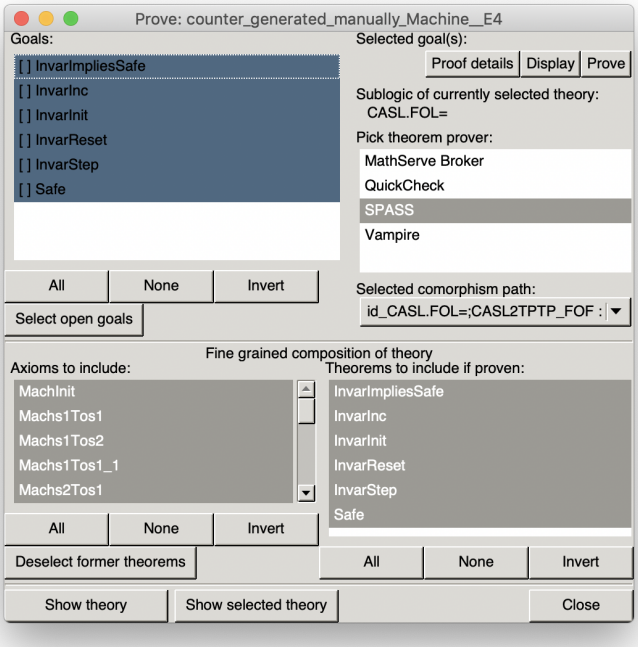


Figure 4.6: The GUI for configuring proof attempts in HETS.

CASL structuring concepts from the Basic Specifications. However, in addition to renaming, we

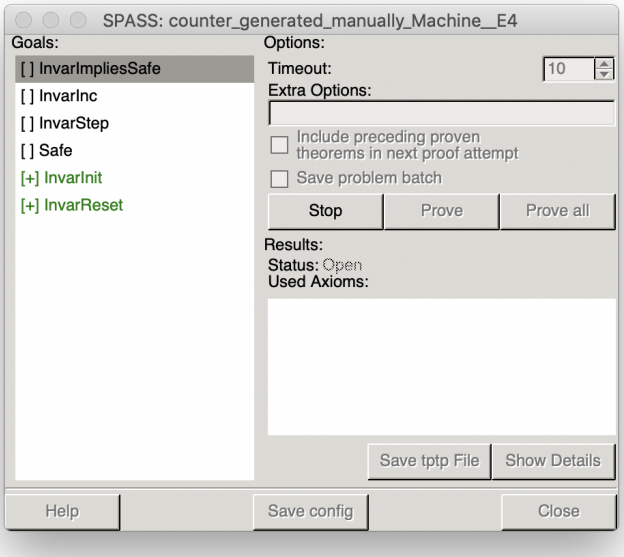


Figure 4.7: A proof window in HETS with some proof obligations resolved (in green) and some open.

now have translation along comorphisms as another kind of change of notation. Specifications can be mixed by translating a specification into the logic of another with which we wish to combine it. Heterogeneous libraries can be written in the CASL extension HETCASL [Mos05] or the language DOL [Mos16], derived from HETCASL and standardised by the OMG. Moreover, HETS can support other notations for structuring concepts, as it does for the Haskell module system in the context of higher-order specifications with HasCASL, another CASL extension.

Returning for a moment to the graph representation of libraries, we see that the nodes are Basic Specifications or specification extensions. The edges show that a specification is based on another, be it through translation from a different logic or through CASL structuring concepts. Some such edges, most importantly those corresponding to uses of **`then %implies`** (e.g., Fig. 4.1, line 39) give rise to proof obligations.

HETS offers a graphical and a textual user interface for discharging such proof obligations.

The GUI interaction for a typical proof could go as follows: Figure 4.3 shows a development graph with named nodes for name specifications and unnamed nodes for intermediate extension steps (separated in HETCASL by the uses of **then**). Applying a rule from the heterogeneous proof system transforms this into the graph of Fig. 4.4. Here proof obligations have been turned into local proof obligations in a node (in red) in one logic. We will later deal with cases where proof obligations are localised into several nodes. Figure 4.5 shows the actions we can perform on the red node containing the proof obligation. Choosing the action *Proof* leads us to the window of Fig. 4.6, where we can configure our proof attempt. In particular, we can choose a prover, a comorphism into the input language of that prover, as well as a selection of axioms and

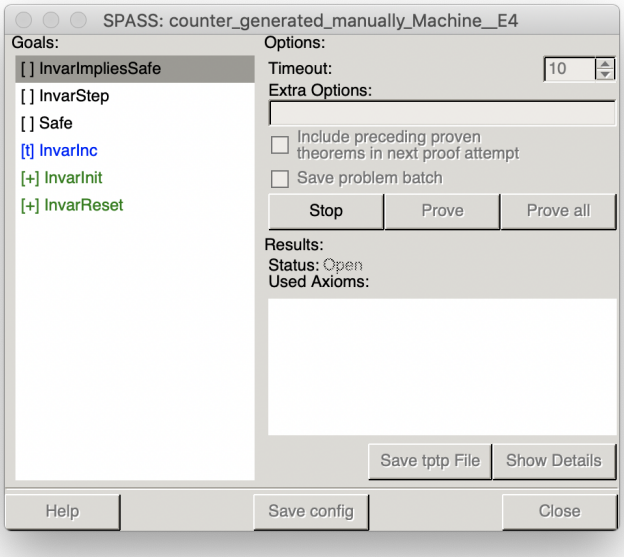


Figure 4.8: A proof window in HETS with some proof obligations resolved (in green), one timed out (in blue), and some open.

proof goals to include. The interaction with the chosen prover then occurs through the window shown in Fig. 4.7. The selected prover here is SPASS [WDF+09], a fully automatic prover, so interaction is restricted to starting and stopping proof attempts with variations in timeout and other options.

4.3 A Short Introduction to Haskell

We will briefly introduce the reader to the three main aspects setting Haskell apart from most earlier functional programming languages, namely typeclasses, lazy evaluation and encapsulation of effects using monads. This section and the next were written as a Jupyter notebook[3] with IHaskell[4]. This allows us to automatically display results of evaluation, type inference, etc.

4.3.1 Typeclasses

A typeclass can be thought of as a predicate with one or more types[5,6] as arguments, together with certain values (the methods), the types of which can mention the typeclass arguments. The

[3] `https://jupyter.org/`

[4] `https://github.com/gibiansky/IHaskell`

[5] They are type level functions, which are also commonly called types in this context.

[6] Some typeclass features, such as typeclasses with multiple parameters, require language extensions. Several such extensions are in common use.

type expressions for the methods have to involve some of the argument types of the typeclass. The predicate is defined loosely: It is is always possible to add an instance for another combination of argument types.

This enables us to write programs with a constrained parametric polymorphism[7]: Expressions which are given solely in terms of the methods of certain type classes are available for all combinations of types implementing those type classes.

A simple example is the functor typeclass:

```
[1]: class Functor f where
       fmap :: (a -> b) -> f a -> f b
```

`fmap` then has the constrained polymorphic type:

```
[2]: :type fmap
```

```
fmap :: forall (f :: * -> *) a b. Functor f => (a -> b) -> f a -> f b
```

Instances of this typeclass should be thought of as endofunctors, where objects are Haskell types, and morphisms are Haskell functions. Then the typeclass parameter `f` is the functor's object part and `fmap` is the morphism part. `fmap` promotes a function on so-called pure values to a function on functorial values. Of course, the same value can be thought of as pure or functorial in different contexts. It is common to require the instances of a typeclass to satisfy some laws – for functors, these are the usual preservation laws for identities and composition.

The following defines the `Functor` instance for the "list of" ' endofunctor:

```
[3]: instance Functor []  where
       fmap g []     = []
       fmap g (x:xs) = g x : fmap g xs
```

4.3.2 Lazy Evaluation

Haskell uses non-strict evaluation. In practice, implementations use lazy evaluation, which is non-strict evaluation with sharing. Strictness can be explicitly enabled in certain situations.

Non-strictness here means that Haskell terms are only evaluated as far as is necessary. This usually means that when a function matches one of its arguments against a constructor[8], the argument is evaluated until the constructor at its root is known. Failure to produce such a constructor, be it through non-termination or some other way, is represented by a special value $\perp$, which each Haskell type contains.

For example, the following defines a nonterminating computation by a simple recursion:

```
[4]: regress = regress
```

[7] Unconstrained parametric polymorphism is also possible.

[8] This is the situation for Algebraic Data Types, "Int", "Char", etc., . For values of function types, λ-abstraction is treated somewhat like a constructor.

The evaluation of `regress` would then fail to terminate. The follwing likewise produces $\perp$, but through an incomplete case distinction:

[5]:
```
incompleteHead (x:xs) = x
```

[6]:
```
incompleteHead []
```

```
<interactive>:1:1-25: Non-exhaustive patterns in function␣
↪incompleteHead
```

Another way of producing $\perp$ is by explicitly raising an error:

[7]:
```
error "Oops..."
```

```
Oops...
CallStack (from HasCallStack):
  error, called at <interactive>:1:1 in interactive:Ghci117
```

However, non-strictness allows us to usefully define:

[8]:
```
aList          = [1,regress]
anInfiniteList = [1,3..]
sharingFun xs  = incompleteHead xs + length xs
```

Now the following computations all yield results:

[9]:
```
incompleteHead aList
```

```
1
```

[10]:
```
anInfiniteList !! 7
```

```
15
```

[11]:
```
sharingFun aList
```

```
3
```

In words, we can take the head of a list with undefined second element. We can likewise take a (sufficiently defined) infinite list and retrieve its seventh entry. Finally, we see that Haskell can compute the length of a list with an undefined element, since the length of a list only depends on its spine, formed by the constructors `(:)` (i.e., cons) and `[]` (i.e., empty list).

In this setting, a function argument has to be a representation of the computation that would produce both the constructor and the representation of the constructor's arguments. Haskell implementations avoid copying these representations, but instead pass them around by means of pointers, thus also avoiding the need to evaluate them repeatedly, which could arise in a naive implementation of non-strict semantics. For example, without this sharing aspect of lazy evluation, the argument to `sharingFun` would be copied and (partly) evaluated twice.

Sharing is an implementation detail, which does not show up in the denotational semantics and is not required by the Haskell standards. In practice, however, Haskell implementations use lazy evaluation, i.e., non-strict evaluation with sharing.

The theoretical underpinnings for strict and non-strict evaluation and their denotational semantics are to be found in domain theory [Sco82].

4.3.3 `Monads`: Encapsulating Side Effects

In Haskell, a function should be thought of as doing nothing beyond producing a value of its return type. This does not mean that effectful programs cannot be implemented in Haskell, only that their type must[9] indicate which effects are possible.

A `Monad` is a special kind of `Functor`. Its type level function turns a type of values into a type of so-called monadic actions with result values of the original type. `Monad`s must implement two methods, the first being:

```
return :: Monad m => a -> m a
```

This method takes a pure value and turns it into a computation that does nothing beyond returning that value. In other words, it allows us to consider a pure computation as an effectful one.

The second method of `Monad` is:

```
(>>=)  :: Monad m => m a -> (a -> m b) -> m b
```

This method is pronounced "bind" and is, roughly, sequential composition of two actions. However, the second action can depend on the result of the first, so the actual second argument is a family of actions, as can be seen in the type signature.

These two methods are expected to satisfy:

1) the left identity law:

```
return x >>= f == f x
```

2) the right identity law:

[9] There are ways around this, which we will not discuss here.

```
m >>= return == m
```

3) associativity:

```
(m >>= f) >>= g == m >>= (\x -> f x >>= g)
```

A `Monad` can be defined simply by implementing an instance of the `Monad` typeclass. Haskell has a special notation for writing monadic computations in an imperative style without losing the effect encapsulation:

[12]:
```
import Control.Monad
divisors :: (Integral a, Enum a, Show a) => a -> [String]
divisors bound = do
  x <- [2..bound]
  y <- [2..x]
  guard (x `mod` y == 0)
  return (show y ++ " divides " ++ show x)
divisors 4
```

```
["2 divides 2","3 divides 3","2 divides 4","4 divides 4"]
```

The above is a computation in the list monad, which can be considered to provide a non-determinism effect. Sequencing in the do-notation is equivalent to ordinary applications of (>>=), where the intermediate results are λ-bound.

In terms of list comprehensions, the above computation could be stated as:

[13]:
```
divisors bound = [ show y ++ " divides " ++ show x
                 | x <- [2..bound]
                 , y <- [2..x]
                 , x `mod` y == 0
                 ]
divisors 4
```

```
["2 divides 2","3 divides 3","2 divides 4","4 divides 4"]
```

Haskell provides an `IO-Monad`, which contains most effects one would usually expect from an imperative programming language, including File IO, mutable variables and concurrency. Beyond this, there are many specialised `Monad`s, to which, as mentioned, the user can easily add. One of the areas where such monads have proved very successful is parsing, to which we will turn our attention in the next section.

4.4 Parsec

The parsers of the various input languages of HETS are implemented using Parsec [LM01]. Parsec is a library of monadic parser combinators for Haskell and, to our knowledge, was the first industrial strength parser combinator library. As of 13th of June 2022, 1023 packages on Hackage[10] depend on parsec[11] and a further 907 on the attoparsec[12] library built on similar principles.

Parser combinators, in contrast to parser generators like yacc, are directly implemented as functions in the host language.

We will show the use of Parsec by means of the classic `expr` language from [ALSU07]. To do so, we first recapitulate its grammar in the form of Haskell type declarations. The grammar defines the problem which the parser then solves.

The particular language we are dealing with consists of simple arithmetical expressions without variables, with the usual operator precedence rules.

[14]:
```
data EXPR   = EXPR :+ TERM    | EXPR :- TERM
            | TermExpr TERM
            deriving Show
data TERM   = TERM :* FACTOR  | TERM :/ FACTOR
            | FacTerm FACTOR
            deriving Show
data FACTOR = DigitFac  DIGIT | ParensFac EXPR
            deriving Show
type DIGIT  = Char -- ['0'..'9']
```

The above are (the parse tree types of) the nonterminals of the `expr` language. The constructors *(:+)*, *(:-)*, *(:*)* and *(:/)* encode the basic arithmetic operators. Next, we will bring relevant modules into scope and define the type for our parsers:

[15]:
```
import Text.Parsec
import Data.Functor ( ($>) )
type Parser t = Parsec String () t
```

In Parsec, a parser can parse some type of streams (here: `String`), keeping state of some type (here: `()`, so we are not keeping any state) and give a result of some type (the type variable `t`, as we have several different parse tree types).

Now, it is time to see the first parser:

[16]:
```
digitP :: Parser DIGIT
digitP = oneOf ['0'..'9'] <?> "DIGIT"
```

Here we used two parser combinators: `oneOf` and `(<?>)`

[10] Hackage, the main Haskell package archive site, is accessible at `https://hackage.haskell.org/`.

[11] Hackage reverse dependencies of parsec according to `https://packdeps.haskellers.com/reverse/parsec`.

[12] Hackage reverse dependencies of attoparsec according to `https://packdeps.haskellers.com/reverse/attoparsec`.

Given any list `cs` of characters, `oneOf cs` consumes the first character `c` and, if `c `elem` cs`, returns it. If the character does not match, an error occurs:

[17]:
```
parseTest (oneOf ['0'..'9']) "5a"
```

```
'5'
```

[18]:
```
parseTest (oneOf ['0'..'9']) "a5"
```

```
parse error at (line 1, column 1):
unexpected "a"
```

The combinator `(<?>)` takes a parser `p` and a string `s`, describing the unit we want to parse. The parser `p <?> s` behaves just like `p` in the case of success, but uses `s` to give more meaningful error messages in the case of parse failures:

[19]:
```
parseTest digitP "a5"
```

```
parse error at (line 1, column 1):
unexpected "a"
expecting DIGIT
```

Parsers can have values of any type as their result, including functions. We will make use of this in the following auxiliary definitions:

[20]:
```
exprOpP :: Parser (EXPR -> TERM -> EXPR)
exprOpP = do char '+' $> (:+)
      <|> do char '-' $> (:-)
termOpP :: Parser (TERM -> FACTOR -> TERM)
termOpP = do char '*' $> (:*)
      <|> do char '/' $> (:/)
pairP :: Parser a -> Parser b -> Parser (a,b)
pairP aP bP = do a <- aP
                 b <- bP
                 return (a,b)
op :: a -> (a->b->c, b) -> c
op a (f,b) = f a b
```

Here, we combine alternative parsers with `(<|>)`. Parsec tries to parse the first option. If that fails without consuming any input, the second one is tried. If, however, the first parser does consume input, we are committed to that option. This behaviour is due to the ineffeciencies that would arise if Parsec always preserved the information required for backtracking. However, controlled backtracking can be introduced throught the `try` combinator: Where `p` would fail after consuming input, `try p` will fail without consuming any.

We have seen two more new combinators: `char` has the obvious meaning. The `($>)` combinator combines a parser `p` with a pure value `v`. The resulting parser returns `v` if `p` succeeds.

The remaining parsers refer to each other, so we present them together:

[21]:
```
factorP :: Parser FACTOR
factorP = (     do DigitFac  <$> digitP
            <|> do ParensFac <$>
                       (char '(' *> exprP <* char ')')
          ) <?> "FACTOR"
exprP :: Parser EXPR
exprP = ( do term1    <- TermExpr <$> termP
             opsTerms <- many $ pairP exprOpP termP
             return $ foldl op term1 opsTerms
        ) <?> "EXPR"
termP :: Parser TERM
termP = ( do fac1    <- FacTerm <$> factorP
             opsFacs <- many $ pairP termOpP factorP
             return $ foldl op fac1 opsFacs
        ) <?> "TERM"
parseTest factorP "7*5+8"
```

```
DigitFac '7'
```

[22]:
```
parseTest factorP "(7*5+9)"
```

```
ParensFac (TermExpr (FacTerm (DigitFac '7') :* DigitFac '5') :+␣
 ↪FacTerm (DigitFac '9'))
```

New combinators used here are:

– `(<$>)`: This is the same as fmap and is available for all functors. In the case of parsers, `f <$> p` parses with `p` and applies `f` to the result.

– `many`: Parses zero or more matches of its argument parser, collecting the results in a list.

– `(*>)` and `(<*)`: Each of these combines two parsers, ignoring the result of the first (`(*>)`) or the second (`(<*)`).

This should be sufficient to aquaint the reader with the ideas underlying Parsec and to handle many common parsing problems. The complete API reference of Parsec is available at `https://hackage.haskell.org/package/parsec`.

Chapter 5

Related Work

Contents

We now turn to a comparison of our work with other approaches to the formalisation of and verification of properties for parts of the UML. The approaches we found fall into the categories of fully automatic model-checking on the one hand, and the use of interactive theorem provers on the other hand. In contrast to the model-checkers, the provers can be applied where infinitely many states have to be considered. However, this is bought with a partial loss of automation.

For each approach, after a general description and a comparison with our approach, we summarise one or more examples which the authors of the approach have used to illustrate the same. These example summaries are not meant to make separate points about how each approach compares to ours, but rather to support the reader's imagination in reflecting the goals, methods and potential of each approach.

A number of other authors give formal semantics to communicating State Machines, however with a purpose different from symbolic analysis of UML. Most authoritatively, the Object Management Group provides an executable semantics of UML Composite Structures [Obj19]. Their objective is to provide an interpreter for the executable subset fUML of the UML. However the approaches we discuss in more detail here are all focused on symbolic analysis.

Closely related to our approach are the works [KMRG15, KM17]. Both these papers address the topic of communicating state machines, however, both fail to provide entirely correct institutions of State Machines. Learning from the reason for this shortcoming, rather than capturing UML state machines directly as an institution, in Chapter 7 we build up a new logic in which UML State Machines can be embedded. In Chapter 9, we extend this logic for communication. In particular, we treat UML event pools as part of Composite Structure Diagrams rather than of State Machines. State Machines are seen as a completely open system, which is (partially) closed by 'wiring up' in a communication structure. Overall, this leads to a separation of concerns: event pools and transitions can be analysed independently. Chapter 7 contains additional details on how our work is related to theirs, so we will not dwell on this here.

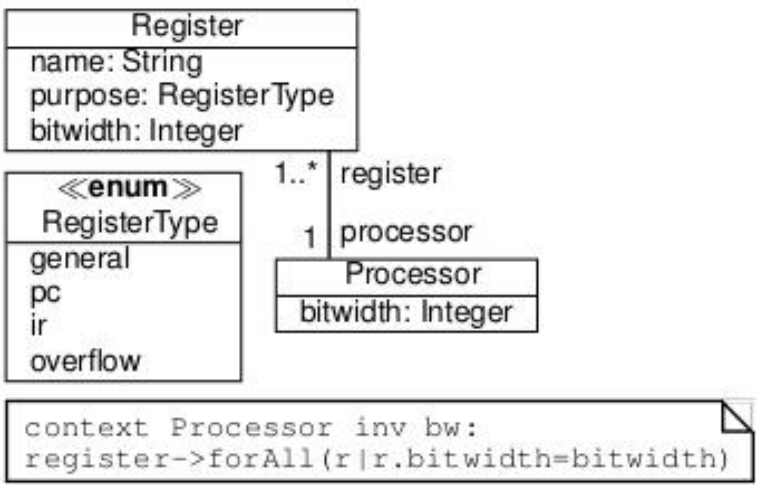


Figure 5.1: A Class Diagram including an OCL constraint from [SWK+10], specifying a processor with its registers.

Next, we will discuss several approaches that fall into the two categories we outlined in a bit more detail.

5.1 Automatic Verification of Finite UML Models

First, let us consider two examples of the fully automatic approaches, one using SAT-solving, the other using the UMC model-checker.

5.1.1 "Verifying UML/OCL Models Using Boolean Satisfiability"

Soeken et al. [SWK+10] use SAT solving to tackle a class of UML verification problems. Given a Class Diagram, a set of OCL constraints, and finite bounds on the number of objects and on the domains of attributes, they generate propositional formulae. SAT solving on these formulae can then determine whether a satisfying instantiation exists and whether the OCL constraints are free of redundancy. The authors report having implemented this approach in a C++ tool. They have performed benchmarks, in which they report obtaining better verification times than the existing tools USE and Alloy in those cases where a comparison was possible.

We see the main advantage of this approach in the highly tuned tools which are available for SAT solving. In terms of structure, their treatment is more general than ours, in that we only treat the fixed structures described in Composite Structure diagrams, whereas they treat Class Diagrams, in which associations can admit a range of cardinalities, and the number of total instances is not fixed. However, their approach is only concerned with structure, whereas our focus is on behaviour. Moreover, the theory of institutions allows our approach to be integrated with other parts of UML and other specification mechanisms.

As an example, Soeken et al., discuss a verification problem over the Class Diagram in Fig. 5.1. The diagram specifies a processor in relationship to its registers, for both of which classes are given. Further, the diagram contains an enumeration type used for encoding the

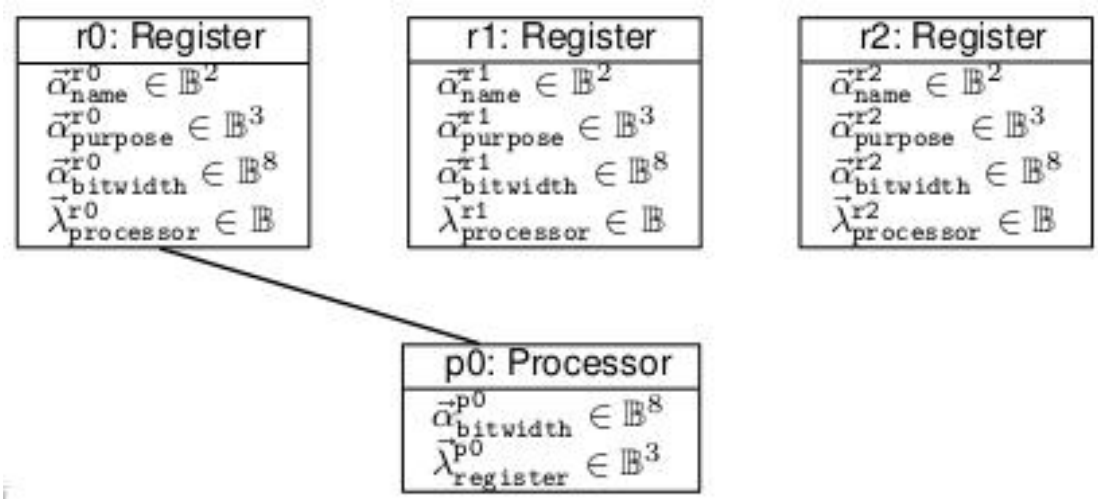


Figure 5.2: A diagram from [SWK+10], showing the SAT-encoding of a system state which could be checked for conformance to Fig. 5.1.

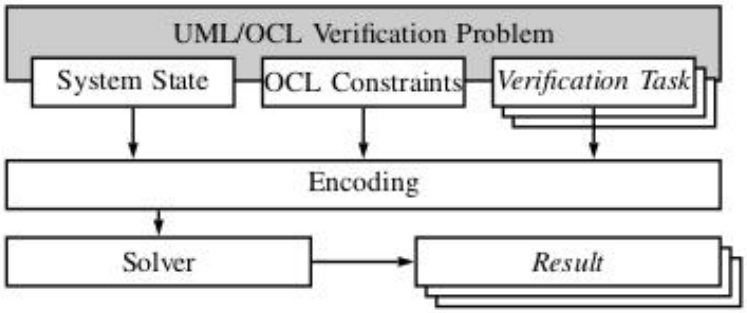


Figure 5.3: A diagram from [SWK+10], showing their verification flow.

purpose of a register. Multiplicities require each register to be linked to exactly one processor and each processor to one or more registers. Attributes specify, among other things, a bitwidth for each register and for each processor. An OCL constraint requires the bitwidths of a processor's registers to equal that of the processor itself.

Figure 5.2 shows a finite system state over this Class Diagram. It further shows how that system is encoded into propositional variables, i.e., variables ranging over bits: Each link (represented by a $\vec{\lambda}$ value) between objects and each attribute (represented by an $\vec{\alpha}$ value) is represented by, in this case, one, three, or eight bits. For the links, each bit encodes the presence or absence of that link to one of the appropriately typed objects. For the attributes, strings are here abstracted to a 2 bit representation, a register's purpose uses three bits for the four cases of the enum and the special value `null`. The integers representing bitwidth are here restricted to eight bits each.

Soeken et al. can then express properties over system state by propositional logic formulae.

For example, they express the OCL constraint from Fig. 5.1 as

$$\bigwedge_{i=0}^{|\mathrm{oid}(\texttt{Register})|-1} \left[\vec{\lambda}^{\texttt{p0}}_{\texttt{register}}[i] \Rightarrow \left(\vec{\alpha}^{\texttt{ri}}_{\texttt{bitwidth}} = \vec{\alpha}^{\texttt{p0}}_{\texttt{bitwidth}}\right)\right].$$

This replaces the `forAll` over a known finite number of registers by the conjunction of a finite family of propositional formulae. Each register is only considered if it is linked to the one processor. Then, the `bitwidth` attributes are required to agree – the equation is shorthand for the obvious propositional equivalences between corresponding bits. In like manner, the Class Diagram itself is encoded into propositional formulae.

SAT solving over these then yields information about the specification's consistency: If the SAT solver returns a satisfying assignment of the propositional variables, this corresponds to a valid system state, and the specification is consistent. If the formulae are reported to be unsatisfiable, there is no valid system state conforming to the bounds imposed, but the Class Diagram and OCL constraints may or may not be consistent. If SAT solving does not complete within acceptable resource bounds, no information is gained, of course.

Verification tasks other than consistency can likewise be checked using a SAT solver. This requires encoding the verification task as another propositional formula, included in the SAT problem given to the solver – the verification flow is shown in Fig. 5.3. The authors report verifying the independence of certain OCL constraints from certain models, where a constraint is considered independent if it is not implied by the requirements of the model itself.

5.1.2 "From EU Projects to a Family of Model Checkers"

UMC[1] by terBeek et al. [tBGM15] is a modelling language and associated model checker meant to correspond to a subset of UML. It is part of the KandISTI suite for on-the-fly model-checking. UMC provides a language for specifying classes, each with an associated State Machine, a statically determined set of objects instantiating those classes and a set of abstraction rules. The State Machines can be hierarchical. The abstraction rules determine which aspects of the system should be visible during model-checking. Each system so specified is closed off from the outside world in that no messages can enter or leave it. UMC then provides a fully automatic tool to check its temporal logic UCTL against such models.

UMC supports more features of UML State Machines than do we. However, for the features we do support, our approach has mainly one advantage in principle and one in terms of tools: It is somewhat more general in principle in that we show properties that a State Machine – or a system of such – has in all possible contexts, whereas UMC checks properties in one particular context. This one context can of course contain a nondeterministic machine modelling an almost arbitrary environment. However, the nondeterminism only applies to control states, and therefore, to finitary branching. Branching between arbitrary values from an infinite data type is not possible in an approach based on state enumeration. In terms of tools, our approach makes several, ideally all formal methods tools available to establish a property – this could include UMC in those cases to which it is well suited.

[1] An online interface to UMC can be accessed at `http://fmt.isti.cnr.it/umc/V4.8/umc.html`

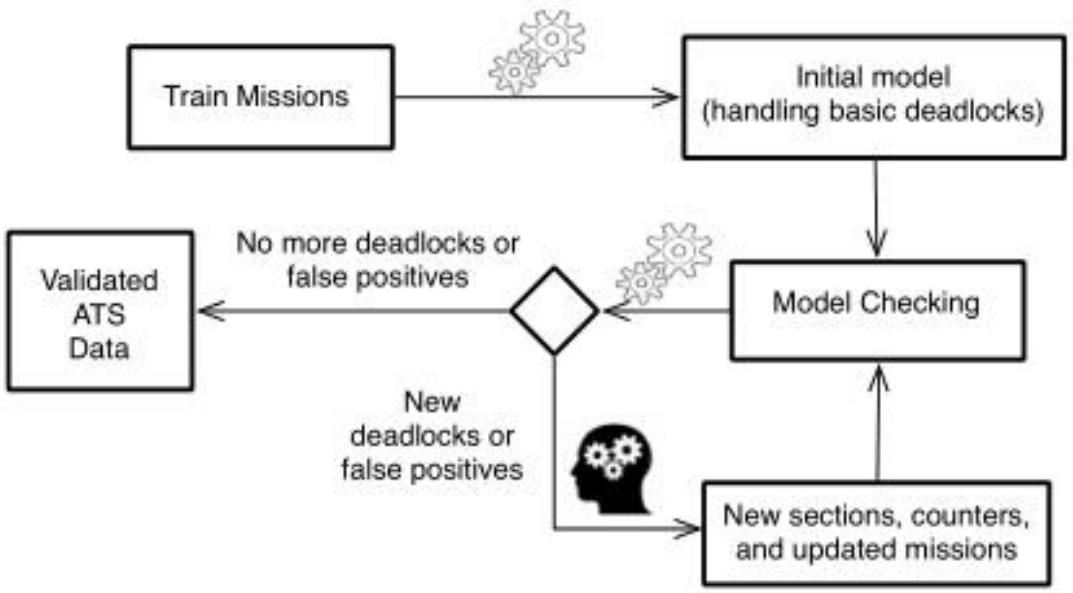


Figure 5.4: Figure from [MSLF14], showing their tool-supported process for generating a model of an Automatic Train Supervision system.

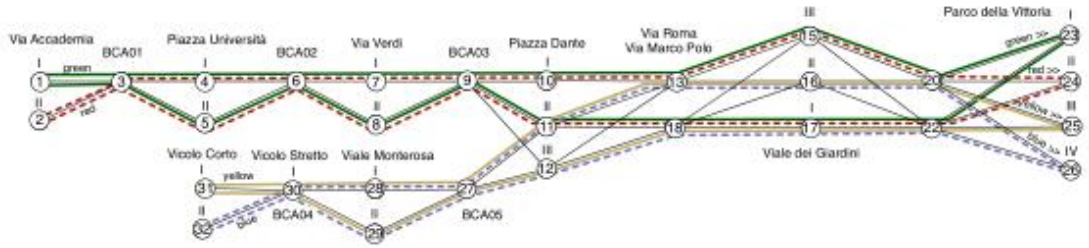


Figure 5.5: A metro network layout from [MSLF14].

One case-study to which UMC has been applied is the prevention of deadlocks in railway routing. We will base the remainder of this section on a paper reporting on this case study, namely [MSLF14] by Mazzanti et al, wherein they provide a tool-supported process (Fig. 5.4) for modelling an Automatic Train Supervision system (ATS). The ATS routes metro trains between stations, ensuring that each of the trains will eventually arrive at its final destination. This implies that the ATS has to prevent deadlock situations. Deadlocks arise when trains block each other, which translates straightforwardly into a classical resource-based formulation of deadlocks: Trains that move acquire and release track segments, where the segments are considered not at the signalling level, but somewhat more abstractly. A deadlock then exists *iff* all movements of trains from their current position would require the previous release of some track segment.

The approach starts with a discrete-time model given by a collection of trains. Each train has as its state a current point and a mission, i.e., a list of points it ought to traverse. The missions of

all trains form the overall metro network layout. An example layout can be seen in Fig. 5.5. At this modelling stage, the system can make a transition *iff* there is a train whose next mission point is unoccupied. The further modelling process then essentially consists in finding further conditions that prevent deadlocks from arising.

To enable the formulation of such conditions, the authors give criteria for identifying basic and then composite critical sections. For each such section, they extend the system state with a counter for the trains present in that section and another variable indicating its capacity. Deadlocks in basic sections arise by trains moving in opposite directions or by a cyclic section's being completely filled with trains, none of which can leave the cycle.

The model is updated with conditions preventing these deadlocks and with actions updating the counters. Figure 5.6 shows an example of a possible resulting transition rule. Then, model-checking is used to detect two conditions which necessitate another iteration:

1. There are still deadlocks, so some more composite critical sections must be identified. This situation is identified by defining the abstraction rule in Fig. 5.7, defining what it means for all trains to have arrived, and then checking whether such a state is always finally reached, i.e., whether the CTL-like[2] UMC formula

   ```
   AF ARRIVED
   ```

   holds.

2. There are forbidden transitions that would not necessarily lead to deadlocks (false positives). Such transitions can arise due to the way composite critical sections are identified. Eliminating the false positives requires refining the use of counters. No precise algorithm for this refinement is given, but examples of resulting UMC files are provided, and the process of finding false positives is outlined as follows: False positives are found by

   a) removing the conditions for entering critical sections from the model, but

   b) introducing an abstraction `DEAD` for the newly allowed states and

   c) establishing by model-checking that the formula

      ```
      not EF (DEAD and EF ARRIVED)
      ```

      holds, i.e., there is no reachable state that is marked as `DEAD`, but allows the eventual arrival of all trains.

Mazzanti et al. [MSLF14] further discuss how to make the sizes of state spaces more manageable by checking different regions of a layout separately.

[2] CTL[CES86] (Computation Tree Logic) is an established branching-time temporal logic, of which UMC's UCTL is a variant.

```
01:  s1 -> s1
02:   { - [(G1P <13)  and  --  13 is the length of the mission for green1
03:     (G1M[G1P+1] /= R1M[R1P]) and                          ----
04:        (G1M[G1P+1] /= G2M[G2P]) and                        |
05:          (G1M[G1P+1] /= R2M[R2P]) and                      |
06:        (SA + G1C[G1P+1][0] <= MAXSA)  and                  |
07:          (SB + G1C[G1P+1][1] <= MAXSB)  and                | Guard
08:            (SC + G1C[G1P+1][2] <= MAXSC)  and              |
09:              (SD + G1C[G1P+1][3] <= MAXSD)  and            |
10:                (SE + G1C[G1P+1][4] <= MAXSE)  and          |
11:                 (SF + G1C[G1P+1][5] <= MAXSF)] /     ----
12:      SA := SA + G1C[G1P+1][0];      ----
13:      SB := SB + G1C[G1P+1][1];          |
14:      SC := SC + G1C[G1P+1][2];          |
15:      SD := SD + G1C[G1P+1][3];          | Actions
16:      SE := SE + G1C[G1P+1][4];          |
17:      SF := SF + G1C[G1P+1][5];          |
18:      G1P := G1P +1;                ----
19:    }
```

Figure 5.6: A state transition for the train `Green1` from [MSLF14]. The array `G1M` is the train's mission. `G1P` is the train's position, given as an index into the mission. Missions from other trains are mentioned to forbid collisions. `SA` to `SF` are train counters for critical sections. The two-dimensional array `G1C` (an example defined in Fig. 5.8 prescribes how movements of `Green1` affect these counters. Each critical section counter also has a prescribed maximum value.

```
Abstractions  {
State SYS.G1P=13  and
      SYS.G2P=13  and
      SYS.R1P=13  and
      SYS.R2P=13  ->  ARRIVED
}
```

Figure 5.7: An abstraction rule from [MSLF14] for detecting deadlocks. For a system with two green trains and two red ones, the abstract name `ARRIVED` is given to system states where each train's position is the final one in its mission (here 13 for all).

```
G1C: int[] :=  -- Section counters updates to be performed by train Green1
 --A,B,C,D,E,F
 [[1,0,0,0,1,0],      --1          [0,0,0,0,0,0],      --92-91
 [-1,0,1,0,0,1],      --1-3        [0,0,0,1,0,0],      --11-9
 [0,0,0,0,0,0],       --3-4        [0,0,0,0,0,0],      --9-8
 [0,0,-1,1,-1,-1],    --4-6        [0,0,1,-1,1,1],     --8-6
 [0,0,0,0,0,0],       --6-7        [0,0,0,0,0,-1],     --6-5
 [0,0,0,-1,0,0],      --7-9        [1,0,-1,0,0,0],     --5-3
 [0,0,0,0,0,0],       --9-92       [0,0,0,0,0,0]];     --3-1
```

Figure 5.8: A definition from [MSLF14] of how each movement in the mission of the train `Green1` affects the train counters for critical sections `A` to `Z`, given in the form of a two-dimensional array.

5.2 Interactive Verification of Potentially Infinite Models

Having seen two examples of fully automatic UML verification approaches, we now give three examples of interactive approaches: First we discuss an approach using the PVS prover; then a second one, which builds a notion of system models within the Isabelle theorem prover; and finally, an approach based on symbolic execution and on the KIV prover.

5.2.1 "Formalising UML Models and OCL Constraints in PVS"

Kyas et al. [KFdB+05] introduce a translation of certain Class Diagrams, of non-hierarchical State Machines and of OCL constraints into the input language of the PVS theorem prover. Then they use PVS to interactively prove a safety property formulated in OCL. As intermediate steps, they formulate invariants directly in PVS and prove them.

The approach is similar to ours in many ways. Verification is done by first translating into a conventional logic of an existing prover. Again, their treatment includes Class Diagrams and OCL constraints and is therefore more general than ours in terms of structure. In terms of behaviour it covers a similar subset of UML State Machine features. At the intersection of structure and behaviour their approach, unlike ours, covers object creation. Being a symbolic approach like ours, it can deal with properties that involve infinitely many configurations. Like us, they encountered the difficulties of reasoning about message queues.

Our approach goes beyond theirs in the following ways: We give principles of inductive proof steps with invariants that could be automatically generated (see Chapter 13). This will, of course, never cover all proofs and lemmas for all verification tasks. Where it is possible for our translations, it will usually also be possible for theirs if one makes use of the advances in proof automation and computation power that have been made since 2005. A further advantage of our approach is, again, the multiplicity of tools we have available for proving. But already in terms of specification itself we profit from the modularity provided by Institution Theory. The authors of [KFdB+05] in giving a joint formalisation for Class Diagrams, State Machines and OCL, cannot just compose existing formalisations of the subformalisms. Extending their approach to a larger fragment of the UML suffers from similar problems. Our approach, on the other hand, can more easily integrate formalisations of other subformalisms of UML, as well as unrelated formalisms, as long as the formalisms are institutions and suitable comorphisms are given.

Kyas et al. show the application of their approach to the example of the Sieve of Eratosthenes, a classical technique for generating prime numbers. They verify the property that the generated numbers are actually primes.

Figure 5.9 shows their architecture, consisting of the classes `Generator` and `Sieve`. There are associations from `Generator` to `Sieve` and from `Sieve` to `Sieve`, which are both called `itsSieve`.

The system contains exactly one `Generator`, the behaviour of which follows Fig. 5.10. It starts by creating the first `Sieve`. The generator keeps a counter `x`, which must at first be initialised to 2. Every 20 time units, the value of `x` is forwarded to the first `Sieve`, after which `x` is incremented.

Each `Sieve` behaves in accordance with Fig. 5.11. A `Sieve` repeatedly receives numbers. The first number so received must be prime and is stored in the attribute `p`. After that, a successor

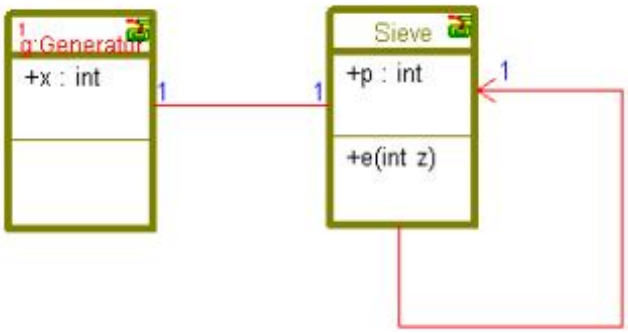


Figure 5.9: A Class Diagram from [KFdB+05], showing the architecture of their example system, an implementation of the Sieve of Eratosthenes, a classical technique for generating prime numbers.

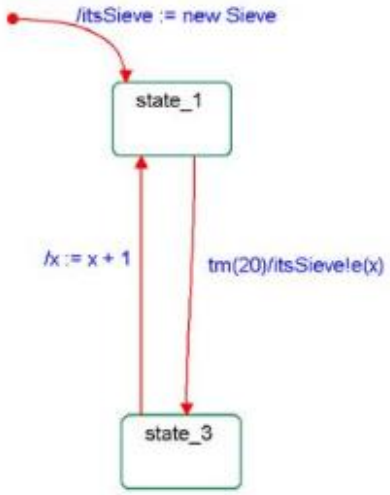


Figure 5.10: A State Machine diagram from [KFdB+05] for the `Generator` class from Fig. 5.9.

`Sieve` is created, and each further number that is received is tested for divisibility by `p`. If it is divisible, it is silently dropped, otherwise it is forwarded to the next `Sieve`. In this manner, a pipeline of `Sieves` is constructed, corresponding to a linked list of all primes less than `x`. If `x` is passed through to the last `Sieve` in the pipeline, it is not divisible by any smaller prime and, therefore, is prime itself. The authors express this safety property by the following OCL constraint:

```
context Sieve inv: Integer{2..(p-1)}->forAll(i | p.mod(i) <> 0)
```

In words: In any `Sieve`, `p` only takes values divisible by no number from 2 to $\mathtt{p} - 1$.

The authors have developed a compiler that generates the specification corresponding to the diagrams in PVS, and likewise the verification goal. Figure 5.12 shows the result of translating a transition from Fig. 5.11 into PVS. In particular, it shows definitions for

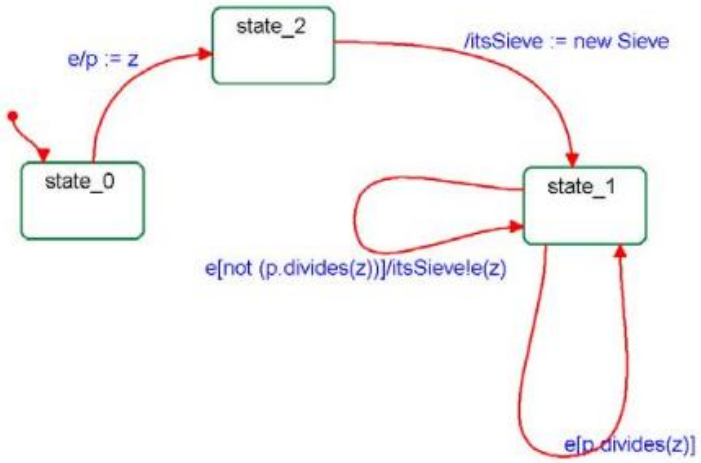


Figure 5.11: A State Machine diagram from [KFdB+05] for the `Sieve` class from Fig. 5.9.

```
Location: TYPE+ = { Generator___61,
  Generator___64, Generator___66,
  Sieve___25, Sieve___28,
  Sieve___32, Sieve___33 }
t28: Transition = (#
  source := Sieve___28,
  trigger := signalEvent(Sieve__e, Sieve__z),
  guard := (LAMBDA (val:
    Valuation): (NOT
      divides((val`aval(Sieve__p)),
    (val`aval(Sieve__z)))))),
  actions := (cons((emitSignal(Sieve__itsSieve,
    Sieve__e, (LAMBDA (val: Valuation):
      (val`aval(Sieve__z))))), null)),
  target := Sieve___28,
  class := Sieve
#);
transitions: setof[Transition] = { t: Transition |
  t = t9 OR t = t11 OR t = t13 OR
  t = t25 OR t = t28 OR t = t32 OR
  t = t34 OR t = t37 }
```

Figure 5.12: A specification fragment in PVS from [KFdB+05], showing the result of translating a transition of the `Generator` State Machine from Fig. 5.10, namely, the transition going from `state_1` to `state_1` and passing a number to the next `Sieve`.

– a `TYPE+` (i.e., a non-empty type), giving the locations (i.e., control states) for both State Machines by an enumeration of nullary constructors;

– the `Transition` itself, given as a record of a `source`, a `trigger`, a `guard`, a list of `actions`, a `target`, and the `class` to which the transition belongs, whereof the `guard` is of function type, making use of PVS's higher-order capabilities; and

– a `setof[Transition]`, stating which transitions are available in the whole system of a `Generator` and `Sieve`s.

The goal turns out to depend on particular properties of the event queues. After formulating and proving these as lemmas, the authors were able to prove the safety property.

5.2.2 "System-model Based Definition of Object-Based Modelling Languages with Semantic Variation Points" (title translated by us)

Grönniger [Grö10] defines a notion of model for object based systems (called a system model). A model (in the sense of UML) in a UML sublanguage is then interpreted to select a set of conforming system models. The author further proposes a general approach to treating variation points in the concrete and abstract syntax, as well as the semantics of modelling languages. Grönninger applies this approach to variation points of the UML. He provides a translation of several UML sublanguages to Isabelle/HOL, and a formalisation of the theory of system models in the same language, thus providing a proof tool.

Compared to our approach, the main theoretical difference of the system model approach is its complexity, due to the need to treat all aspects of all possible object based systems. Their treatment is modular in the sense that it is given by structured theories, but if at any point a variation on the notion of system model itself became necessary, this would lead to new, not easily comparable, model classes for existing UML models. Properties once thought established would be subject to doubt again. Likewise, translations into a complex formalism involve many possibilities of making bad choices, which likewise would be hard to undo without losing established results.

In terms of tools, the system model is formalised in Isabelle/HOL. The interactive Isabelle system contains integrations of several automatic provers. Isabelle/HOL is also integrated into HETS, allowing to some extent the highly principled tool reuse and formalism combination that Institution Theory permits, although with some difficulties: Reuse of existing tools for the system model formalism would require comorphisms out of higher-order logic. For these comorphisms to exist, the target logic would need to have signatures and sentences corresponding to *all* signatures and sentences of higher-order logic. Going in the other direction, translations from another formalism into Isabelle/HOL would not usually result in sentences over the system model signatures, requiring significant formalisation effort within Isabelle to establish a relationship. The somewhat arbitrary nature of this formalised relationship would then impair the usefulness of the guarantees Institution Theory gives us.

That said, the system model approach gives a useful semantics to a significant subset of UML, which makes a powerful tool available to the modeller and provides an integrated view of what an object based system can be. That view, even if we should find it too complex to

ABC — Klassendiagramm

```
1  classdiagram ABC {
2
3    class A;
4    class B extends A;
5    class C extends B;
6  }
```

CA — Klassendiagramm

```
1  classdiagram CA {
2
3    class C;
4    class A extends C;
5  }
```

Figure 5.13: Two Class Diagrams from [Grö10] (given only textually) which, when combined, specify cyclical inheritance.

CDKonf — Konfiguration

```
1  package conf.mc.umlp.cd;
2
3  config CDConf for mc.umlp.cd.CD {
4
5    CDSyntax.LangParam.InvariantExpression.OCLInvariant,
6    CDSyntax.LangParam.Value.JavaExpression,
7    CDSyntax.LangParam.Body.JavaBlockStatement,
8    CDSyntax.Stereos.SingletonClass,
9    CDSyntax.CDConstr.RestrictCD,
10
11   CDSemantics.vStereotypes.MapSingletonClass,
12   CDSemantics.vMapSuperClasses.MapSuperclassesDirect,
13   CDSemantics.vMapMethod.MapMethod,
14   CDSemantics.vMapConstructor.MapConstructor,
15   CDSemantics.vMapModifiers.MapModifiers,
16   CDSemantics.vMapAssociations.MapAssociations,
17
18   SystemModel.vObject.AntiSymSub,
19   SystemModel.vControl.TypeSafeOps,
20   SystemModel.vData.QualifiedAssociation,
21   SystemModel.vData.CompositeAssociation,
22   SystemModel.vData.DerivedAssociation,
23   SystemModel.vData.BinaryAssociation,
24   SystemModel.vData.StaticAttr,
25   SystemModel.vControl.StaticOpn,
26   SystemModel.vState.Query
27 }
```

Figure 5.14: A selection of language features from [Grö10].

connect to formally, can still inspire our approach to modelling parts of such a system and the relationships between such models.

We will now discuss three example systems verified by [Grö10] using the system model approach.

The first such verification task refers to the Class Diagrams shown in Fig. 5.13. The inheritance relationships specified in the two diagrams constitute a cycle. The author then shows, for a particular language variant, that this cycle makes the combination of these two diagrams inconsistent. As discussed, Grönninger [Grö10] studies language variation points. Choices with respect to such variation points can be combined in a language configuration file, which can however leave some options unspecified. An example of such a file is given in Fig. 5.14.

SubNonCirc — Isabelle-Theorie

```
lemma SubNonCirc :
 "⟦valid sm; sub sm C2 C1; sub sm C1 C2⟧ ⟹ C1 = C2"
apply (unfold valid-def)
apply (unfold valid-AntiSymSub-def)
by auto
```

SubTrans — Isabelle-Theorie

```
lemma SubTrans :
  "⟦valid sm; sub sm C1 C2; sub sm C2 C3⟧ ⟹ sub sm C1 C3"
apply (unfold valid-def)
apply (unfold valid-base.simps)
apply (unfold valid-Subclassing.simps)
apply clarify
apply (unfold pSubTrans-def)
by best
```

Figure 5.15: Two Isabelle lemmas with proofs from [Grö10], the first stating that subclass relationships in a valid system model are non-circular, the second stating that they are transitive.

ABC-CA-circ — Isabelle-Theorie

```
lemma ABC-CA-circ :
 "{ sm | sm . valid sm ∧ mCDDefinition ABC sm} ∩
       { sm | sm . valid sm ∧ mCDDefinition CA sm} = {}"
apply auto
(* unfold definitions *)
apply (unfold mSuperClass-def)
apply simp
apply (unfold mNameList-def)
apply (frule SubTrans)
apply auto
apply (frule SubNonCirc)
apply auto
back
done
```

Figure 5.16: The Isabelle theorem from [Grö10] establishing the incompatibility, in a particular language configuration, of the Class Diagrams from Fig. 5.13. Some proof steps about the unfolding of definitions have been skipped.

This configuration selects, among others, the variant `AntiSymSub`, which requires the subclass relation to be antisymmetric. The verification task at hand can only be carried out if this variant is active.

The configuration and the diagrams are translated to Isabelle. The diagram languages are deeply embedded into Isabelle, so the diagrams are translated into an explicit representation of their syntax. Based on the configuration, there is an axiomatisation for a semantic function, which is likewise represented explicitly in Isabelle as a thing to be reasoned about. The lemmas shown in Fig. 5.15 then establish that, in a valid system model, subclass relationships cannot be cyclical and must be transitive. The theorem shown in Fig. 5.16 then concludes that the two diagrams from Fig. 5.13 are in fact incompatible, i.e., that the sets of system models satisfying each have an empty intersection.

Calendar

Klassendiagramm

```
classdiagram Calendar {

  class DateTime {
    int date;
    int time;
    boolean before(DateTime d) {
      return date < d.date
           || date == d.date && time < d.time;
    }
  }

  class Entry {
    DateTime start;
    DateTime end;
  }

  [
    context Entry e inv: e.start.before(e.end);
  ];

}
```

Figure 5.17: The Class Diagram for the calendar from [Grö10].

Calendar1

Objektdiagramm

```
objectdiagram Calendar1 {

  e : Entry {
    start = d1;
    end   = d2;
  }

  d1 : DateTime {
    date = 3;
    time = 1;
  }

  d2 : DateTime {
    date = 2;
    time = 2;
  }

}
```

Figure 5.18: The Object Diagram for the calendar from [Grö10].

As a second example, the author discusses the modelling of a calendar. First, the general concept of a calendar is modelled in a Class Diagram (Fig. 5.17). The Class Diagram models points in time via the `DateTime` class and then, using these, models calendar entries (class: `Entry`), where a calendar entry is given by a start time and an end time. Further, an Object Diagram is given (Fig. 5.18). The author then shows a lemma (Fig. 5.19) which restates the invariant from the Class Diagram and proves that this holds in all reachable states of system models conforming to the Class Diagram. Another lemma (Fig. 5.20) is shown, which restates

InvariantHolds — Isabelle-Theorie

```
lemma InvariantHolds: "⟦mCDDefinition Calendar sm⟧ ⟹
     ∀s∈reachable sm.
       Oid ''e'' ∈ oids s ∧
       VOid (Oid ''e'') ∈ CAR sm (TO (Class ''Entry'')) ⟶
         startDate (Oid ''e'') s  < endDate (Oid ''e'') s ∨
         startDate (Oid ''e'') s = endDate (Oid ''e'') s ∧
         startTime (Oid ''e'') s < endTime (Oid ''e'') s"
(* unfold definitions *)
by auto
```

Figure 5.19: The lemma from [Grö10] restating the invariant from the Class Diagram Fig. 5.17.

AttrValues — Isabelle-Theorie

```
lemma AttrValues :
   "⟦s ∈ reachable sm; mODDefinition Calendar1 s sm⟧ ⟹
     Oid ''e'' ∈ oids s ∧
     VOid (Oid ''e'') ∈ CAR sm (TO (Class ''Entry'')) ∧
     startDate (Oid ''e'') s = 3 ∧
     startTime (Oid ''e'') s = 1 ∧
     endDate (Oid ''e'') s =  2 ∧
     endTime (Oid ''e'') s  = 2 "
(* unfold definitions *)
by auto
```

Figure 5.20: The lemma from [Grö10] restating the attribute values from the Object Diagram Fig. 5.18.

InvariantViolated — Isabelle-Theorie

```
lemma InvariantViolated :
   "mCDDefinition Calendar sm ⟶
        (∀ s ∈ reachable sm . ¬ mODDefinition Calendar1 s sm)"
apply auto
apply (frule InvariantHolds)
apply (frule AttrValues)
apply auto
apply (erule ballE)
by auto
```

Figure 5.21: The theorem from [Grö10] showing that the objects from Fig. 5.18 violate the constraint from Fig. 5.17.

the content of the Object Diagram. Finally, the author proves that the Object Diagram violates the Class Diagram (Fig. 5.21).

As a third example, an authentication system is discussed, which is specified by the State Diagram in Fig. 5.22. The Sequence Diagram Fig. 5.23 then shows a run where the system authenticates a user who offered an incorrect key. After several lemmas, which we do not show here, the author proves the theorem Fig. 5.24, which shows that the State Chart rules out the run from the Sequence Diagram.

Auth — Statechart

```
1  package mysys;
2
3  statechart Auth Auth  {
4
5    initial state NonAuth;
6
7    state Auth;
8
9    NonAuth -> NonAuth : [key != ownkey] auth(key) / {return false;}
10
11   NonAuth -> Auth : [key == ownkey] auth(key) / {return true;}
12
13 }
```

Figure 5.22: The State Diagram for the authentication system from [Grö10].

Auth1 — Sequenzdiagramm

```
1  package mysys;
2
3  <<match="visible">> sequencediagram Auth1  {
4
5    u:User;
6    a:Auth;
7
8    u -> a : auth(someKey);
9    <a : [someKey != ownkey]>;
10   u <- a : return true;
11
12 }
```

Figure 5.23: The Sequence Diagram from [Grö10] showing a hypothetical (illegal) run of the authentication system specified in Fig. 5.22.

IllegalMsg — Isabelle-Theorie

```
1  lemma IllegalMsg :
2    "valid sm ∧
3     traceProp sm trc ∧
4     mSCDefinition Auth sm
5       ⟶ ¬ mSDDefinition Auth1 trc sm"
6  apply (unfold traceProp-def)
7  (* unfold definitions and instantiate quantifiers *)
8  by auto
```

Figure 5.24: A theorem from [Grö10] stating that the Sequence Diagram Fig. 5.22 violates the State Diagram Fig. 5.23.

5.2.3 "Interactive Verification of UML State Machines"

Balser et al. [BBK+04] provide a calculus for temporal logic into which they embed UML State Machines. They implement their calculus in the interactive theorem prover KIV. The tool Hugo/RT provides a translation of XMI files into the temporal logic, thus connecting their approach to any full-featured UML modelling tool on the input side. They then use KIV to

verify properties stated in the same temporal logic.

Our approach is a direct successor of theirs. It results from the ongoing effort to find formalisations and translations of State Machines that satisfy the coherence conditions of Institution Theory. We found a difficulty in principle of such approaches in our Master's thesis [Ros17] and developed a modal logic of State Machines up to refinement, with an embedding of fully refined State Machines into that logic. The logic we formalise and implement in this thesis for State Machines is a development on the logic from our Master's thesis. More on the underlying problems and our approach to solving them can be found in Chapter 7.

A main advantage from [BBK+04] for the user of their proof tool is their use of symbolic execution. In future work, we may well want to integrate a similar feature into HETS, either for generating lemmas supporting fully automatic verification, or for discharging certain proof obligations by that mechanism itself.

We also make use of KIV to verify an example of a Composite Structure, however, we only use KIV's first-order and inductive capabilities, and only in a fully automated fashion.

In terms of examples, Balser et al. prove a property about an ATM scenario, an example family that is often found in the literature. We ourselves have performed proofs on a similar example, which we will discuss in Sect. 13.2. Their form of the example contains one class, `Bank`, for which a Class Diagram and State Machine are shown in Fig. 5.25[3]. Over this system, they want to prove that, in all reachable configurations with control state `DispenseMoney`, it holds that the maximum number of tries has not been exceeded and the flag `PINValid` has been set. They express this through the formula:

$$\text{(prop)}\ \ \Box(\text{DispenseMoney} \rightarrow \text{tries} \leq \text{maxTries} \wedge \text{PINValid})$$

Figure 5.26 shows the flow of information by which the authors transform such a State Machine into a form usable for proving: First, a suitable UML editor is used to create the diagrams. The editor must be able to export the diagrams in the XMI format. The tool Hugo/RT, developed by the authors of the paper, can then be used to translate from XMI into a KIV specification, from which desired properties can then be proved.

For the example at hand, after fixing an initial condition and an invariant, Balser et al. arrive at the proof graph shown in Fig. 5.27. It is worth stressing that the proof graph is not a tree because states that can be reached by several paths are nevertheless treated only once – the authors refer to this as sequencing. The structure of the proof graph is closely linked to the possible runs of the State Machine. Each proof step is either directly the execution of a State Chart step, or it expresses sequencing, or the application of the inductive hypothesis. The numbers on the nodes of the proof tree refer back to the states shown in Fig. 5.25 – a close inspection of this correspondence will do much to support the reader's intuition for how the symbolic execution supports the proof development.

[3] In particular, this means that unlike us, they do not separate the ATM from the bank, because their focus is on the semantics of a single State Machine.

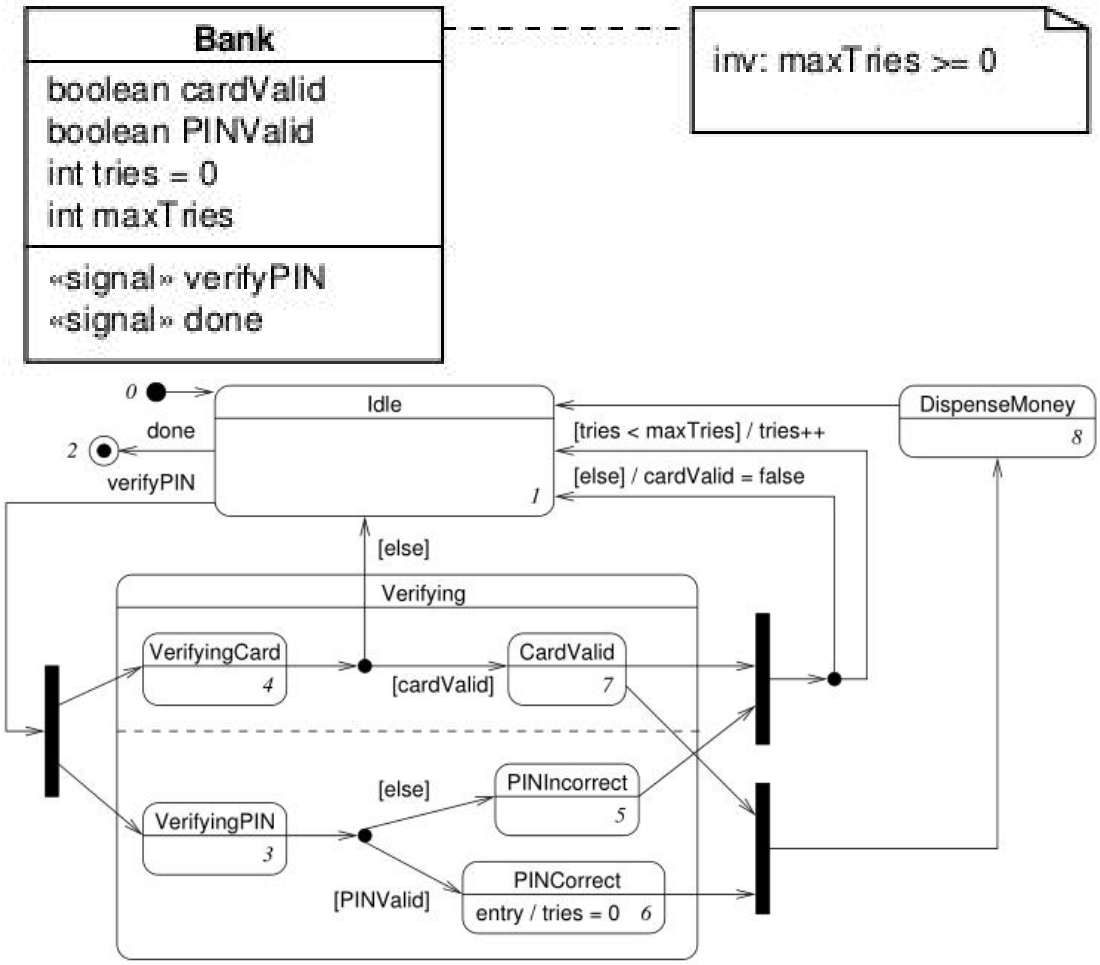


Figure 5.25: The Class Diagram and State Machine for the Bank example from [BBK+04].

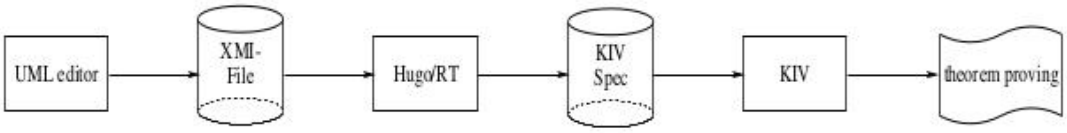


Figure 5.26: A diagram from [BBK+04] showing their verification flow.

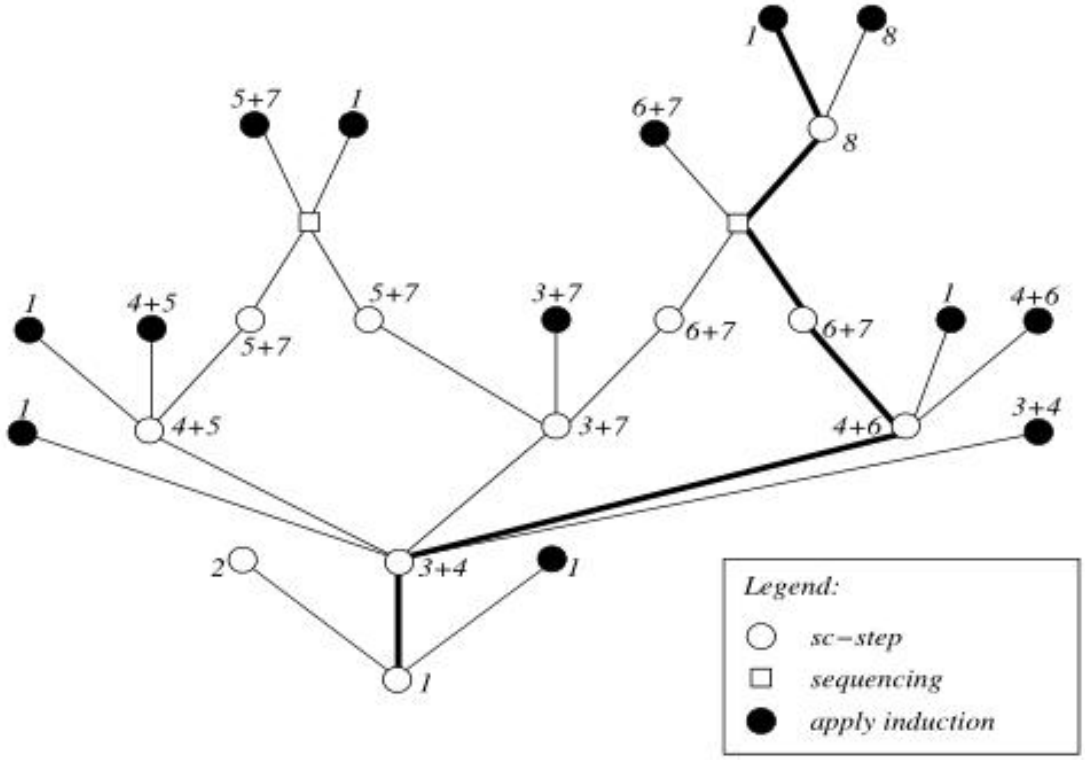


Figure 5.27: The proof tree for the example from [BBK$^+$04].

Part II

Methodology Construction

Chapter 6

A Bird's-eye View of our Languages

In this chapter, we give an overview over the input languages, logics and translations developed or used within this thesis project and our two-stage process for developing them.

The most important points in this respect are summarised in Fig. 6.1: Arranged vertically, we have several formal logics, which are established or conjectured to be institutions, with (theoroidal) comorphisms translating between them. This starts at the top with our institution for UML Composite Structures. Then, we have our modal logic for event/data systems in two variants: EDHML (Event/Data Hybrid Modal Logic, without output events) and EDHMLO (with output events). Composite Structures translate into the latter variant. From our modal logics, we can then translate into CASL, the central logic of the HETS framework, in which we are working. From there, we have translations to the tool-facing logics. Of these, we have used SoftFOL via an automatic and KIV via a manual translation. Other tool-facing languages could be added here. SoftFOL and the fragment of KIV we use (first-order logic with inductive data types) are standard enough that a violation of the Satisfaction Conditions for institutions or comorphisms seems unlikely here. To our knowledge, however, these results have never been formally established, so they remain conjectural.

Our input languages somewhat depart from the institutional logics. In the Composite Structures institution, which is built over EDHMLO, we have merely factored out the underlying State Machines. In the case of the State Machines, there is a larger difference: There are technical difficulties to directly creating a language for State Machines that is an institution[1]. Therefore, we needed to create a more expressive modal logic, which is a bit further removed from the structure of machines themselves. Sentences in this logic demand the presence or absence of certain transitions and could be regarded as expressing machines up to refinement. There is then an algorithm for constructing a formula that requires exactly the behaviour of one particular State Machine. This algorithm corresponds to the semantic function in Fig. 6.1, from outside the institutional world (depicted as coming from the side rather than above), and embeds State Machines (with or without outputs) into the relevant modal logic institution.

As mentioned before, we developed our languages in two steps. The first step treats State Machines in isolation, i.e., they can consume inputs, but their actions include no communication.

[1] For more detail on these difficulties and their consequences for our approach, see Chapter 7.

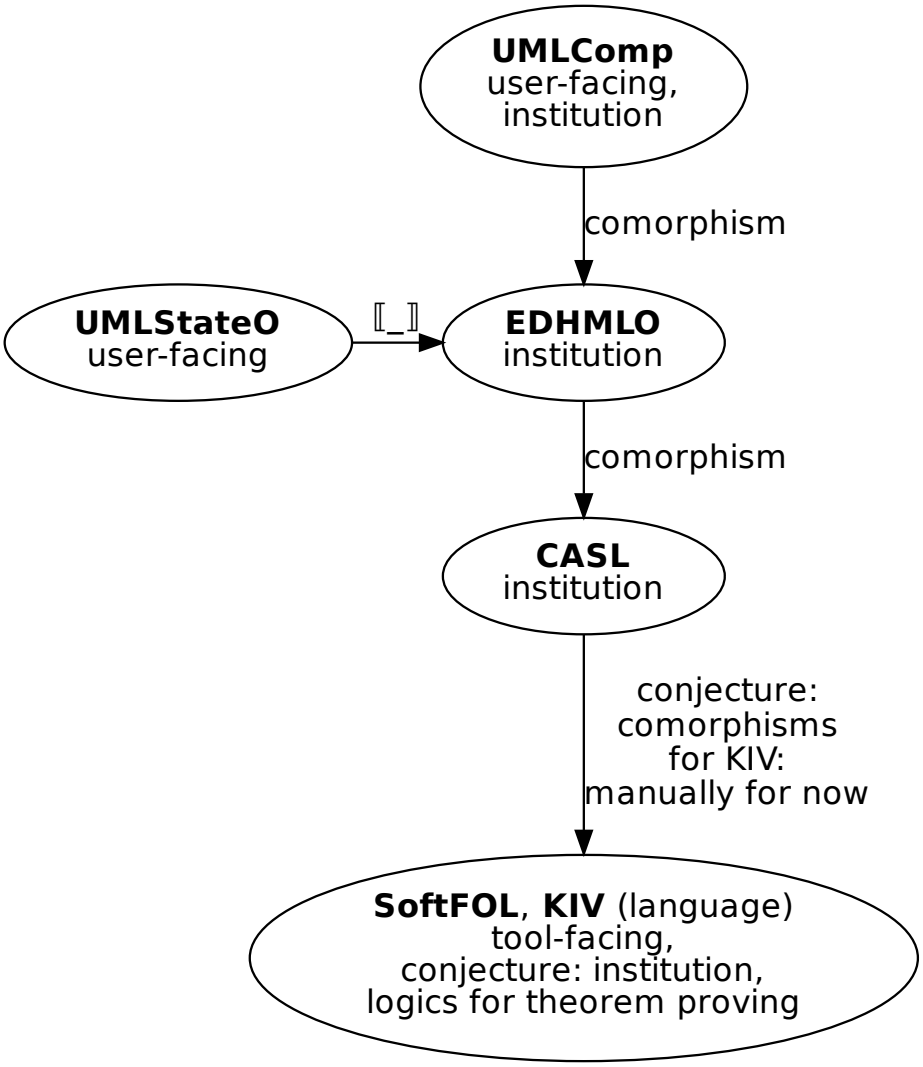


Figure 6.1: The languages and formalisms involved in our verification approach.

Actions here are limited to updating the data of the local configuration. For these machines, we have developed the input language UMLSTATE and the institution EDHML. Chapter 7 explains EDHML, as well as why and how we embed State Machines into it. Appendix A contains the grammars (abstract and concrete syntax, static semantics) for our concrete State Machine input language UMLSTATE. Chapter 8 introduces the theoroidal comorphism from EDHML to CASL.

In a second step, we developed an extended notion of State Machines, suitable for combination

in Composite Structures, on which we then built a notion of Composite Structures themselves. Our extended State Machine languages UMLStateO and EDHMLO are distinguished from the isolated versions mainly by the addition of output events and, in a mostly transparent way, of ports. We discuss these languages mathematically in Chapter 9 and the comorphism to CASL in Chapter 10. Chapter 11 outlines mathematically how an institution for Composite Structures can be built over EDHMLO. A comorphism to CASL is obtained with the help of a flattening into EDHMLO. In Chapter 12 we discuss our surface languages and their implementation in HETS. Chapter 13 contains some verification experiments based on our approach.

Chapter 7

EDHML: An Institution for Embedding Simple State Machines

Contents

In this chapter, we motivate and present an institution for the modal logic *Event/Data Hybrid Modal Logic (*EDHML*)*, which allows us to embed Simple UML State Machines. As discussed in Chapter 3, modal logics are used to reason about notions of possibility and necessity. A hybrid modal logic additionally has constructs to require a sentence to hold in particular states/possible worlds. Events and data encode the notions of communication and state from UML which we wish to capture.

In our EDHML the relevant notion of possibility is that of a machine allowing a particular kind of transition. The modality is a specification of the events consumed in such a transition. The formula under the modality makes requirements of the state in which we arrive, which can themselves again be modal formulæ describing further transitions. The reader will observe that our logic does not necessarily fix the exact set of transitions possible in a machine, but gives a loose specification that can allow further refinement. This is by design and is the key ingredient to formula and structure translations that fulfil the Satisfaction Condition. We then describe an algorithm for translating a particular State Machine into that logic.

The contents of this chapter have appeared in similar form as part of our paper at WADT 2020 [RBKR20].

The construction of EDHML was a common effort between Alexander Knapp, Markus Roggenbach and myself. In our virtual meetings, we jointly sketched the constructions, proofs and examples, which I then worked out in detail and wrote down in their final form.

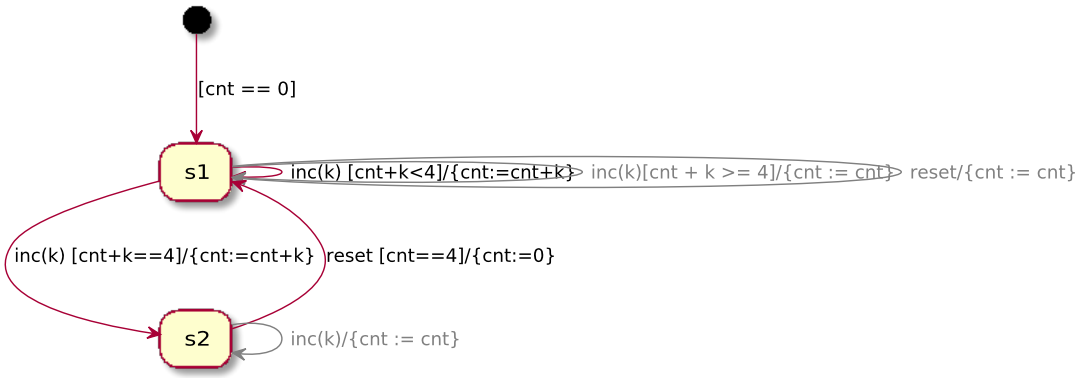


Figure 7.1: Simple UML State Machine *Counter*

. The red transitions are those one would usually show in a UML State Machine. In UML, by default, an event is implicitly discarded if it occurs in a state where it is not explicitly handled. Here, these dicarding transitions are displayed in grey.

7.1 Example: The Counter Machine

Figure 7.1 shows the example of a bounded, resettable counter working on an attribute cnt (assumed to take values in the natural numbers) that is initialised with 0. The counter can be reset to 0 or increased by a natural number k, subject to the current control state (*s1* or *s2*) and the guards (shown in square brackets) and effects (after the slash) of the outgoing transitions. An effect describes how the data state before firing a transition (referred to by unprimed attribute names) relates to the data state after (primed names) in a single predicate; this generalises the more usual sequences of assignments such that $\mathsf{cnt}' = \mathsf{cnt} + k$ corresponds to $\mathsf{cnt} := \mathsf{cnt} + k$ and $\mathsf{cnt}' = \mathsf{cnt}$ to a skip. The machine is specified non-deterministically: If event $\mathsf{inc}(k)$ occurs in state *s1* such that the guard $\mathsf{cnt} + k = 4$ holds, the machine can either stay in *s1* or it can proceed to *s2*. Seemingly, the machine does not react to reset in *s1* and to inc in *s2*. However, UML State Machines are meant to be input-enabled such that all event occurrences to which the machine shows no explicit reacting transition are silently discarded. We show the explicit transitions in red and the implicit ones in grey. Overall, the machine *Counter* shall ensure that cnt never exceeds 4: In fact, we will prove this in Chapter 13.

For example (the reader can follow along in Fig. 7.1), a run could

- start in $s1$ with $cnt = 0$; then
- consume $inc(6)$, leaving us still in $s1$ with $cnt = 0$, as the limits $cnt == 4$ and $cnt > 4$ explicit, red transitions were exceeded; then
- consume $inc(3)$, leaving us still in $s1$, but now with $cnt = 3$; then
- consume $inc(1)$, taking us to $s2$ with $cnt = 4$; then
- consume $inc(2)$, leaving us still in $s2$ with $cnt = 4$ (via the grey self-loop on $s2$)
- consume *reset*, returning to $s1$, now with $cnt = 0$; and, finally

– consume another *reset*, again leaving us at an unchanged $s1$ with $cnt = 0$ via the rightmost grey self-loop on $s1$.

7.2 Event/Data Structures and the Problem with Institutionalising UML State Machines

It is for such simple UML State Machines as the counter in Fig. 7.1 that we want to provide proof support in SPASS, one of the provers integrated into HETS, via an institutional encoding in CASL. The sub-language covers the following fundamental State Machine features: data, states, and (non-deterministic) guarded transitions for reacting to events. However, for the time being, we defer all advanced modelling constructs, like hierarchical states or compound transitions to future work. An extension of UMLSTATE with communication between machines will be presented in the following chapters in the form of the languages UMLSTATEO (State Machines with outputs) and UMLCOMP (Composite Structures).

In the following we first make precise the syntax of the machines by means of event/data signatures, data states and transitions, guards and effects. Then we introduce semantic structures for the machines and define their model class. Syntax and semantics of simple UML State Machines form the basis for their institutionalisation. We thus also introduce event/data signature morphisms and the corresponding sentence translation and structure reducts in order to be able to change the interface of simple UML State Machines.

7.2.1 Event/Data Signatures, Data States and Transitions

We capture the events for a machine in an *event signature* E that consists of a finite set $|E|$ with elements called events and a map $v(E)$ assigning to each $e \in |E|$ a finite set with elements called variables, where we write $e(X)$ for $e \in |E|$ and $v(E)(e) = X$, and also $e(X) \in E$ in this case. For the data state, we use a *data signature* A consisting of a finite set with elements called attributes. An *event/data signature* Σ consists of an event signature $E(\Sigma)$ and a data signature $A(\Sigma)$.

Example 7.1 The event/data signature Σ of the simple UML State Machine in Fig. 7.1 is given by the set of events $|E(\Sigma)| = \{\mathsf{inc}, \mathsf{reset}\}$ with argument variables $v(E(\Sigma))(\mathsf{inc}) = \{k\}$ and $v(E(\Sigma))(\mathsf{reset}) = \emptyset$ such that $\mathsf{inc}(k) \in E(\Sigma)$ and $\mathsf{reset} \in E(\Sigma)$; as well as the data signature $A(\Sigma) = \{\mathsf{cnt}\}$.

For specifying transition guards and effects, we exchange UML's notorious and intricate expression and action languages both syntactically and semantically by a straightforward CASL fragment rendering guards as data state predicates and effects as data transition predicates, which, in the implemented language, take the form of assignments.

We assume given a fixed universe $\mathcal{D}$ with elements called *data values* and a CASL specification Dt with a dedicated sort dt in its signature $\mathrm{Sig}(Dt)$ such that the universe dt^M of every model $M \in \mathrm{Mod}^{\text{CASL}}(Dt)$ is isomorphic to $\mathcal{D}$, i.e., there is a bijection $\iota_{M,dt} : dt^M \cong \mathcal{D}$. This puts at our disposal the open formulæ $\mathscr{F}^{\text{CASL}}_{\mathrm{Sig}(Dt),X}$ over sorted variables $X = (X_s)_{s \in S}$ and their satisfaction

relation $M, \beta \models^{\text{CASL}}_{\text{Sig}(Dt),X} \varphi$ for models $M \in \text{Mod}^{\text{CASL}}(Dt)$, variable valuations $\beta : X \to M$, and formulæ $\varphi \in \mathscr{F}^{\text{CASL}}_{\text{Sig}(Dt),X}$.

Example 7.2 Consider the natural numbers $\mathbb{N}$ as data values $\mathcal{D}$. The CASL specification in Fig. 4.1 characterises $\mathbb{N}$ up to isomorphism as the carrier set of the dedicated sort $dt = Nat$. It specifies an abstract data type with sort *Nat*, operations $+, 0, suc$, and a predicate $<$.

The very simple choice of $\mathcal{D}$ capturing data with only a single sort can, in principle, be replaced by any institutional data modelling language that, for our purposes of a theoroidal institution comorphism (see Chapter 8), is faithfully representable in CASL; one such possibility are UML class diagrams, see [JKMR12].

Data states and guards. A *data state* ω for a data signature A is given by a function $\omega : A \to \mathcal{D}$; in particular, $\Omega(A) = \mathcal{D}^A$ is the set of A-data states. The guards of a machine are *state predicates* in $\mathscr{F}^{\mathcal{D}}_{A,X} = \mathscr{F}^{\text{CASL}}_{\text{Sig}(Dt),A\cup X}$, taking A as well as an additional set X as variables of sort dt. A state predicate $\phi \in \mathscr{F}^{\mathcal{D}}_{A,X}$ is to be interpreted over an A-data state ω and valuation $\beta : X \to \mathcal{D}$ and we define the *satisfaction relation* $\models^{\mathcal{D}}$ by

$$\omega, \beta \models^{\mathcal{D}}_{A,X} \phi \iff M, \iota^{-1}_{M,dt} \circ (\omega \cup \beta) \models^{\text{CASL}}_{\text{Sig}(Dt),A\cup X} \phi$$

where $M \in \text{Mod}^{\text{CASL}}(Dt)$ and $\iota_{M,dt} : M(dt) \cong \mathcal{D}$. For a state predicate $\varphi \in \mathscr{F}^{\mathcal{D}}_{A,\emptyset}$ not involving any variables, we write $\omega \models^{\mathcal{D}}_{A} \varphi$ for $\omega \models^{\mathcal{D}}_{A,\emptyset} \varphi$.

Example 7.3 The guard $\mathsf{cnt} + k \leq 4$ of the machine in Fig. 7.1 features both the attribute cnt and the variable k. A data state fulfilling this state predicate for $k = 0$ is $\mathsf{cnt} \mapsto 3$.

Data transitions and effects. A *data transition* (ω, ω') for a data signature A is a pair of A-data states; in particular, $\Omega^2(A) = (\mathcal{D}^A)^2$ is the set of A-data transitions. It holds that $(\mathcal{D}^A)^2 \cong \mathcal{D}^{2A}$, where $2A = A \uplus A$ and we assume that no attribute in A ends in a prime $'$ and all attributes in the second summand are adorned with an additional prime. The effects of a machine are *transition predicates* in $\mathscr{F}^{2\mathcal{D}}_{A,X} = \mathscr{F}^{\mathcal{D}}_{2A,X}$. The satisfaction relation $\models^{2\mathcal{D}}$ for a transition predicate $\psi \in \mathscr{F}^{2\mathcal{D}}_{A,X}$, data transition $(\omega, \omega') \in \Omega^2(A)$, and valuation $\beta : X \to \mathcal{D}$ is defined as

$$(\omega, \omega'), \beta \models^{2\mathcal{D}}_{A,X} \psi \iff \omega + \omega', \beta \models^{\mathcal{D}}_{2A,X} \psi$$

where $\omega + \omega' \in \Omega(2A)$ with $(\omega + \omega')(a) = \omega(a)$ and $(\omega + \omega')(a') = \omega'(a)$.

Example 7.4 The effect $\mathsf{cnt}' = \mathsf{cnt} + k$ of the machine in Fig. 7.1 describes the increment of the value of attribute cnt by a variable amount k.

7.2.2 Syntax of Simple UML State Machines

A simple UML State Machine U uses an event/data signature $\Sigma(U)$ for its events and attributes and consists of a finite set of *control states* $C(U)$, a finite set of *transition specifications*

$T(U)$ of the form $(c,\phi,e(X),\psi,c')$ with $c,c' \in C(U)$, $e(X) \in E(\Sigma(U))$, a state predicate $\phi \in \mathscr{F}^{\mathcal{D}}_{A(\Sigma(U)),X}$, a transition predicate $\psi \in \mathscr{F}^{2\mathcal{D}}_{A(\Sigma(U)),X}$, an *initial control state* $c_0(U) \in C(U)$, and an *initial state predicate* $\varphi_0(U) \in \mathscr{F}^{\mathcal{D}}_{A(\Sigma(U)),\emptyset}$, such that $C(U)$ is *syntactically reachable*, i.e., for every $c \in C(U) \setminus \{c_0(U)\}$ there are $(c_0(U),\phi_1,e_1(X_1),\psi_1,c_1),\ldots,(c_{n-1},\phi_n,e_n(X_n),\psi_n, c_n) \in T(U)$ with $n > 0$ such that $c_n = c$. Syntactic reachability guarantees initially connected State Machine graphs. This simplifies graph-based algorithms (see Alg. 1).

In our input language, we will use different notations for guards and assignments. Guards we will write in the style $expr1 == expr2$. In assignments, instead of transition equations for attributes, we will write in the style $\mathsf{attr1} := expr$, which will then translate to the primed variant $\mathsf{attr1}' = expr$ in the semantic notation and the translation to CASL.

Example 7.5 The machine in Fig. 7.1 has as its control states $\{s1,s2\}$, as its transition specifications $\{(s1,\mathsf{cnt}+k \leq 4,\mathsf{inc}(k),\mathsf{cnt}' = \mathsf{cnt}+k,s1),(s1,\mathsf{cnt}+k = 4,\mathsf{inc}(k),\mathsf{cnt}' = 4, s2),\ (s2,\ \mathsf{true},\ \mathsf{reset},\ \mathsf{cnt}' = 0,\ s1)\}$, as initial control state $s1$, and as initial state predicate $\mathsf{cnt} = 0$.

7.2.3 Event/Data Structures and Models of Simple UML State Machines

For capturing machines semantically, we use event/data structures that are given over an event/data signature Σ and consist of a transition system of configurations such that all configurations are reachable from its initial configurations. Herein, configurations show a control state, corresponding to machine states, and a data name from which a proper data state over $A(\Sigma)$ can be retrieved by a labelling function. Transitions connect configurations by events from $E(\Sigma)$ with their arguments instantiated by data from $\mathcal{D}$.

Formally, a Σ-*event/data structure* $M = (\Gamma,R,\Gamma_0,\omega)$ over an event/data signature Σ consists of a set of *configurations* $\Gamma \subseteq C \times D$ for some sets of *control states* C and *data names* D, a family of *transition relations* $R = (R_{e(\beta)} \subseteq \Gamma \times \Gamma)_{e(X) \in E(\Sigma),\beta:X\to\mathcal{D}}$, and a non-empty set of *initial configurations* $\Gamma_0 = \{c_0\} \times D_0 \subseteq \Gamma$ with a unique *initial control state* $c_0 \in C$ such that Γ is *reachable* via R, i.e., for all $\gamma \in \Gamma$ there are $\gamma_0 \in \Gamma_0$, $n \geq 0$, $e_1(X_1),\ldots,e_n(X_n) \in E(\Sigma)$, $\beta_1 : X_1 \to \mathcal{D},\ldots,\beta_n : X_n \to \mathcal{D}$, and $(\gamma_i,\gamma_{i+1}) \in R_{e_{i+1}(\beta_{i+1})}$ for all $0 \leq i < n$ with $\gamma_n = \gamma$; and a *data state labelling* $\omega : D \to \Omega(A(\Sigma))$. We write $c(M)(\gamma) = c$ and $\omega(M)(\gamma) = \omega(d)$ for $\gamma = (c,d) \in \Gamma$, $\Gamma(M)$ for Γ, $C(M)$ for $\{c(M)(\gamma) \mid \gamma \in \Gamma(M)\}$, $R(M)$ for R, $\Gamma_0(M)$ for Γ_0, $c_0(M)$ for c_0, and $\Omega_0(M)$ for $\{\omega(M)(\gamma_0) \mid \gamma_0 \in \Gamma_0\}$.

The restriction to reachable transition systems is not strictly necessary and could be replaced by constraining all statements on event/data structures to take into account only their reachable part (see, e.g., Lem. 7.11).

Example 7.6 For an event/data structure for the machine in Fig. 7.1 over its signature Σ in Ex. 7.1 we may choose the control states C as $\{s1,s2\}$, and the data names D as the set $\Omega(A(\Sigma)) = \mathcal{D}^{\{\mathsf{cnt}\}}$. In particular, the data state labelling ω is just the identity function. The only initial configuration is $(s1,\{\mathsf{cnt} \mapsto 0\})$. A possible transition goes from configuration $(s1,\{\mathsf{cnt} \mapsto 2\})$ to configuration $(s2,\{\mathsf{cnt} \mapsto 4\})$ with the instantiated event $\mathsf{inc}(2)$.

A $\Sigma(U)$-event/data structure M is a *model* of a simple UML State Machine U if

$C(M) \supseteq C(U)$ up to a bijective renaming, $c_0(M) = c_0(U)$, $\Omega_0(M) \subseteq \{\omega \in |\Omega(A(\Sigma(U)))| \mid \omega \models^{\mathcal{D}}_{A(\Sigma(U))} \varphi_0(U)\}$, and if the following holds for all $(c,d) \in \Gamma(M)$:

- for all $(c,\phi,e(X),\psi,c') \in T(U)$
  and $\beta : X \to \mathcal{D}$
  with $\omega(M)(d),\beta \models^{\mathcal{D}}_{A(\Sigma(U)),X} \phi$,
  there is a $((c,d),(c',d')) \in R(M)_{e(\beta)}$
  with $(\omega(M)(d),\omega(M)(d')),\beta \models^{2\mathcal{D}}_{A(\Sigma(U)),X} \psi$;

- for all $((c,d),(c',d')) \in R(M)_{e(\beta)}$
  there is either some $(c,\phi,e(X),\psi,c') \in T(U)$
  with $\omega(M)(d),\beta \models^{\mathcal{D}}_{A(\Sigma(U)),X} \phi$
  and $(\omega(M)(d),\omega(M)(d')),\beta \models^{2\mathcal{D}}_{A(\Sigma(U)),X} \psi$,
  or $\omega(M)(d),\beta \not\models^{\mathcal{D}}_{A(\Sigma(U)),X} \bigvee_{(c,\phi,e(X),\psi,c') \in T(U)} \phi$, $c = c'$,
  and $\omega(M)(d) = \omega(M)(d')$.

A model of U thus on the one hand implements each transition prescribed by U, but on the other hand must not show semantic transitions beyond those covered by the specified syntactic transitions. Moreover, it is *input-enabled*, i.e., every event can be consumed in every control state: If no precondition of an explicitly specified transition is satisfied, there is a self-loop which leaves the data state untouched. In fact, input-enabledness, as required by the UML specification [Obj17], can also be rendered as a syntactic transformation making a simple UML State Machine U input-enabled by adding the following set of transition specifications for idling self-loops:

$$\{(c, \neg(\bigvee_{(c,\phi,e(X),\psi,c') \in T(U)} \phi), e(X), 1_{A(\Sigma(U))}, c) \mid c \in C,\ e(X) \in E(\Sigma(U))\}\,.$$

Example 7.7 For the simple UML State Machine in Fig. 7.1 the "grey" transitions correspond to an input-enabledness completion w.r.t. the "red" transitions.

7.2.4 Event/Data Signature Morphisms, Reducts, and Translations

The external interface of a simple UML State Machine is given by events, its internal interface by attributes. Both interfaces, represented as an event/data signature, are susceptible to change in the system development process which is captured by signature morphisms. Such changes must also be reflected in the guards and effects, i.e., data state and transition predicates, by syntactical translations as well as in the interpretation domains by semantical reducts.

A *data signature morphism* from a data signature A to a data signature A' is a function $\alpha : A \to A'$. The α-*reduct* of an A'-data state $\omega' : A' \to \mathcal{D}$ along a data signature morphism $\alpha : A \to A'$ is given by the A-data state $\omega'|\alpha : A \to \mathcal{D}$ with $(\omega'|\alpha)(a) = \omega'(\alpha(a))$ for every $a \in A$; the α-*reduct* of an A'-data transition (ω',ω'') by the A-data transition $(\omega',\omega'')|\alpha = (\omega'|\alpha,\omega''|\alpha)$. The

state predicate translation $\mathscr{F}^{\mathcal{D}}_{\alpha,X} : \mathscr{F}^{\mathcal{D}}_{A,X} \to \mathscr{F}^{\mathcal{D}}_{A',X}$ along a data signature morphism $\alpha : A \to A'$ is given by the CASL-formula translation $\mathscr{F}^{\text{CASL}}_{\text{Sig}(Dt),\alpha\cup 1_X}$ along the substitution $\alpha \cup 1_X$; the *transition predicate translation* $\mathscr{F}^{2\mathcal{D}}_{\alpha,X}$ by $\mathscr{F}^{\mathcal{D}}_{2\alpha,X}$ with $2\alpha : 2A \to 2A'$ defined by $2\alpha(a) = \alpha(a)$ and $2\alpha(a') = \alpha(a)'$. For each of these two reduct-translation-pairs the *satisfaction condition* holds due to the general substitution lemma for CASL:

$$\begin{aligned} \omega'|\alpha,\beta \models^{\mathcal{D}}_{A,X} \phi &\iff \omega',\beta \models^{\mathcal{D}}_{A',X} \mathscr{F}^{\mathcal{D}}_{\alpha,X}(\phi) \\ (\omega',\omega'')|\alpha,\beta \models^{2\mathcal{D}}_{A,X} \psi &\iff (\omega',\omega''),\beta \models^{2\mathcal{D}}_{A',X} \mathscr{F}^{2\mathcal{D}}_{\alpha,X}(\psi) \end{aligned}$$

An *event signature morphism* $\eta : E \to E'$ is a function $\eta : |E| \to |E'|$ such that $v(E)(e) = v(E')(\eta(e))$ for all $e \in |E|$. An *event/data signature morphism* $\sigma : \Sigma \to \Sigma'$ consists of an event signature morphism $E(\sigma) : E(\Sigma) \to E(\Sigma')$ and a data signature morphism $A(\sigma) : A(\Sigma) \to A(\Sigma')$. The *$\sigma$-reduct* of a Σ'-event/data structure M' along σ is the Σ-event/data structure $M'|\sigma$ such that

- the model components $\Gamma(M'|\sigma) \subseteq \Gamma(M')$ and $R(M'|\sigma) = (R(M'|\sigma)_{e(\beta)})_{e(X)\in E(\Sigma),\beta:X\to\mathcal{D}}$ are inductively defined by
  - $\Gamma(M'|\sigma) \supseteq \Gamma_0(M')$ and
  - for all $\gamma',\gamma'' \in \Gamma(M')$, $e(X) \in E(\Sigma)$, and $\beta : X \to \mathcal{D}$, if $\gamma' \in \Gamma(M'|\sigma)$ and $(\gamma',\gamma'') \in R(M')_{E(\sigma)(e)(\beta)}$, then $\gamma'' \in \Gamma(M'|\sigma)$ and $(\gamma',\gamma'') \in R(M'|\sigma)_{e(\beta)}$;
- $\Gamma_0(M'|\sigma) = \Gamma_0(M')$; and
- $\omega(M'|\sigma)(\gamma') = (\omega(M')(\gamma'))|\sigma$ for all $\gamma' \in \Gamma(M'|\sigma)$.

Building a reduct of an event/data-structure does not affect the single configurations, but potentially reduces the set of configurations by restricting the available events, and the data state observable from the data name of a configuration. We denote by $\Gamma^F(M,\gamma)$ and $\Gamma^F(M)$, respectively, the set of configurations of a Σ-event/data structure M that are F-reachable from a configuration $\gamma \in \Gamma(M)$ and from an initial configuration $\gamma_0 \in \Gamma_0(M)$, respectively, with a set of events $F \subseteq E(\Sigma)$ where a $\gamma_n \in \Gamma(M)$ is *F-reachable in M from* a $\gamma_1 \in \Gamma(M)$ if there are $n \geq 1$, $e_2(X_2),\ldots,e_n(X_n) \in F$, $\beta_2 : X_2 \to \mathcal{D},\ldots,\beta_n : X_n \to \mathcal{D}$, and $(\gamma_i,\gamma_{i+1}) \in R(M)_{e_{i+1}(\beta_{i+1})}$ for all $1 \leq i < n$.

Lemma 7.8 Let $\sigma : \Sigma \to \Sigma'$ be an event/data signature morphism, $F \subseteq E(\Sigma)$, and M' a Σ'-event/data structure.

1. For all $\gamma'_1,\gamma'_2 \in \Gamma(M')$, if $\gamma'_1 \in \Gamma(M'|\sigma)$, then $(\gamma'_1,\gamma'_2) \in R(M'|\sigma)_{e(\beta)}$ if, and only if, $(\gamma'_1,\gamma'_2) \in R(M')_{E(\sigma)(e)(\beta)}$.
2. For all $\gamma',\gamma'' \in \Gamma(M')$ such that $\gamma' \in \Gamma(M'|\sigma)$, $\gamma'' \in \Gamma^F(M'|\sigma,\gamma')$ if, and only if, $\gamma'' \in \Gamma^{E(\sigma)(F)}(M',\gamma')$.
3. For all $\gamma' \in \Gamma(M')$, $\gamma' \in \Gamma^F(M'|\sigma)$ if, and only if, $\gamma' \in \Gamma^{E(\sigma)(F)}(M')$.

Proof. (1) This follows directly from the inductive definition of the σ-reduct of Σ'-event/data structures.

(2) Let $\gamma', \gamma'' \in \Gamma(M')$ with $\gamma' \in \Gamma(M'|\sigma)$. By induction, it holds that $\gamma'' \in \Gamma^{E(\sigma)(F)}(M')$ if, and only if, there are $n \geq 0$, $e_1(X_1), \ldots, e_n(X_n) \in F$, and $\beta_1 : X_1 \to \mathcal{D}, \ldots, \beta_n : X_n \to \mathcal{D}$, and $(\gamma'_i, \gamma'_{i+1}) \in R(M')_{E(\sigma)(e_{i+1})(\beta_{i+1})}$ for all $0 \leq i < n$ with $\gamma' = \gamma'_0$ and $\gamma'' = \gamma'_n$. Thus, by (1), since $\gamma' \in \Gamma(M'|\sigma)$, $\gamma'' \in \Gamma^{E(\sigma)(F)}(M')$ if, and only if, $\gamma'' \in \Gamma^F(M'|\sigma)$.

(3) Let $\gamma' \in \Gamma(M')$. By definition it holds that $\gamma' \in \Gamma^{E(\sigma)(F)}(M')$ if, and only if, $\gamma' \in \Gamma^{E(\sigma)(F)}(M', \gamma'_0)$ for some $\gamma'_0 \in \Gamma_0(M')$; if, and only if $\gamma' \in \Gamma^F(M'|\sigma, \gamma'_0)$ for some $\gamma'_0 \in \Gamma(M')$ by (2) since $\gamma'_0 \in \Gamma_0(M'|\sigma)$; if, and only if, $\gamma' \in \Gamma^F(M'|\sigma)$ since $\Gamma_0(M'|\sigma) = \Gamma_0(M')$. □

Example 7.9 Let Σ be as in Ex. 7.1 and Σ_0 the event/data signature with $E(\Sigma_0) = E(\Sigma)$ and $A(\Sigma_0) = \emptyset$. Consider the signature morphism $\sigma : \Sigma_0 \to \Sigma$ as the identity on the events and the trivial embedding on the attributes. Let M be a model of the simple UML State Machine in Fig. 7.1. The syntactic transition $(s1, \mathsf{cnt} + k \leq 4, \mathsf{inc}(k), \mathsf{cnt}' = \mathsf{cnt} + k, s1)$ induces, among others, the two semantic transitions $((s1, d_1), (s1, d_2)), ((s1, d_2), (s1, d_3)) \in R(M)_{\mathsf{inc}(\{k \mapsto 1\})}$ where $\omega(M)(d_i) = \{\mathsf{cnt} \mapsto i\}$ for $1 \leq i \leq 3$. In the reduct $M|\sigma$ we find exactly these two semantic transitions, however, $\omega(M|\sigma)(d_i) = \emptyset$ for all $1 \leq i \leq 3$. This illustrates why we distinguish between data states and data names. With the distinction, we have a bijection between semantic transitions in the reduct and semantic transitions in the original structure. Without the distinction, the two different transitions in M would collapse into one transition only as there is just a single data state $\emptyset$.

Although it is straightforward to define a translation of simple UML State Machines along an event/data signature morphism, the rather restrictive notion of their models prevents the satisfaction condition from holding. In fact, this is already true for previous endeavours to institutionalise UML State Machines as part of this general research programme [KMRG15, KM17]. There, machines themselves were taken to be sentences over signatures comprising both events and states, and the satisfaction relation also required that a model should show exactly the transitions of such a machine sentence. For signature morphisms σ that are not surjective on states, building the reduct could result in fewer states and transitions, which leads to the same violation of the satisfaction condition which we found in [Ros17], shown here in Fig. 7.2.

We therefore propose to make a detour through a more general hybrid modal logic. This logic is directly based on event/data structures and thus close to the domain of State Machines. For forming an institution, its hybrid features allow to avoid control states as part of the signature and its event-based modalities allow to specify both mandatory and forbidden behaviour in a more fine-grained manner. Still, the logic is expressive enough to characterise the model class of a simple UML State Machine syntactically.

7.3 A Hybrid Modal Logic for Event/Data Systems

The logic EDHML is a hybrid modal logic for specifying event/data-based reactive systems and reasoning about them. The EDHML-signatures are the event/data signatures, the EDHML-structures the event/data structures. The modal part of the logic allows to handle transitions

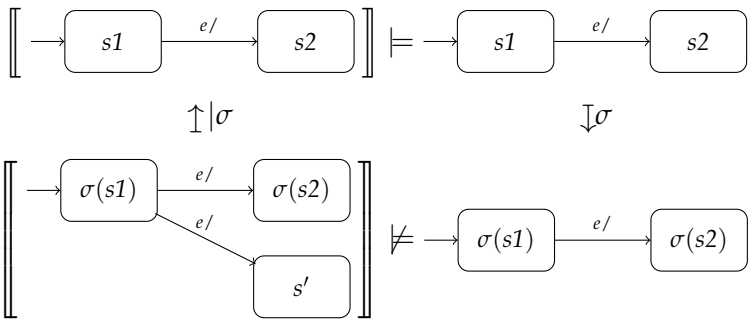


Figure 7.2: A counterexample (assuming s' is not in the image of σ) to the satisfaction condition for earlier attempts at institutionalising UML State Machines.

between configurations where the modalities describe moves between configurations that adhere to a pre-condition or guard as a state predicate for an event with arguments and a transition predicate for the data change corresponding to effects. The hybrid part of the logic allows to bind control states of system configurations and to jump to configurations with such control states explicitly, but leaves out nominals as interfacing names as well as the possibility to quantify over control states.

The logic builds on the hybrid dynamic logic $\mathcal{D}^{\downarrow}$ for specifying reactive systems without data [MBHM16] and its extension $\mathcal{E}^{\downarrow}$ to handle also data [HMK19]. We restrict ourselves to modal operators consisting only of single instead of compound actions as done in dynamic logic. However, we still retain a box modality for accessing all configurations that are reachable from a given configuration. Moreover, we extend $\mathcal{E}^{\downarrow}$ by adding parameters to events.

The category of EDHML-*signatures* $\mathrm{Sig}^{\mathrm{EDHML}}$ consists of the event/data signatures and signature morphisms. The Σ-event/data structures form the discrete category $\mathrm{Mod}^{\mathrm{EDHML}}(\Sigma)$ of EDHML-*structures* over Σ. For each signature morphism $\sigma : \Sigma \to \Sigma'$ in $\mathrm{Sig}^{\mathrm{EDHML}}$ the *σ-reduct functor* $\mathrm{Mod}^{\mathrm{EDHML}}(\sigma) : \mathrm{Mod}^{\mathrm{EDHML}}(\Sigma') \to \mathrm{Mod}^{\mathrm{EDHML}}(\Sigma)$ is given by $\mathrm{Mod}^{\mathrm{EDHML}}(\sigma)(M') = M'|\sigma$. As the next step we introduce the formulæ and sentences of EDHML together with their translation along $\mathrm{Sig}^{\mathrm{EDHML}}$-morphisms and their satisfaction over $\mathrm{Mod}^{\mathrm{EDHML}}$. We then show that for EDHML the satisfaction condition holds and thus obtain EDHML as an institution. Subsequently, we show that EDHML is expressive enough to characterise the model class of simple UML State Machines.

7.3.1 Formulæ and Sentences of EDHML

EDHML-formulæ aim at expressing control and data state properties of configurations as well as accessibility properties of configurations along transitions for particular events. The pure data state part is captured by data state sentences over $\mathcal{D}$. The control state part can be accessed and manipulated by hybrid operators for binding the control state in a state variable, $\downarrow s$; checking for a particular control state, s; and accessing all configurations with a particular control state, $@^F$, which, however, only pertains to reachable configurations relative to a set F of events. Transitions between configurations are covered by different modalities: a box modality for

accessing all configurations that are reachable from a given configuration, $\Box^F$, again relative to a set F of events; a diamond modality for checking that an event with arguments is possible with a particular data state change, $\langle e(X) /\!\!/ \psi \rangle$; and a modality for checking the reaction to an event with arguments according to a pre-condition and a transition predicate, $\langle\!| e(X) : \phi /\!\!/ \psi |\!\rangle$.

Formally, the Σ-*event/data formulæ* $\mathscr{F}^{\mathrm{EDHML}}_{\Sigma,S}$ over an event/data signature Σ and a set of *state variables* S are inductively defined by

- φ — data state sentence $\varphi \in \mathscr{F}^{\mathcal{D}}_{A(\Sigma),\emptyset}$ holds in the current configuration;
- s — the control state of the current configuration is $s \in S$;
- $\downarrow s \,.\, \varrho$ — calling the current control state s, formula $\varrho \in \mathscr{F}^{\mathrm{EDHML}}_{\Sigma,S \uplus \{s\}}$ holds;
- $(@^F s)\varrho$ — in all configurations with control state $s \in S$ that are reachable with events from $F \subseteq E(\Sigma)$, the formula $\varrho \in \mathscr{F}^{\mathrm{EDHML}}_{\Sigma,S}$ holds;
- $\Box^F \varrho$ — in all configurations that are reachable from the current configuration with events from $F \subseteq E(\Sigma)$ formula $\varrho \in \mathscr{F}^{\mathrm{EDHML}}_{\Sigma,S}$ holds;
- $\langle e(X) /\!\!/ \psi \rangle \varrho$ — in the current configuration there is a valuation of X and a transition for event $e(X) \in E(\Sigma)$ with these arguments that satisfies transition formula $\psi \in \mathscr{F}^{2\mathcal{D}}_{A(\Sigma),X}$ and makes $\varrho \in \mathscr{F}^{\mathrm{EDHML}}_{\Sigma,S}$ hold afterwards;
- $\langle\!| e(X) : \phi /\!\!/ \psi |\!\rangle \varrho$ — in the current configuration for all valuations of X satisfying state formula $\phi \in \mathscr{F}^{\mathcal{D}}_{A(\Sigma),X}$ there is a transition for event $e(X) \in E(\Sigma)$ with these arguments that satisfies transition formula $\psi \in \mathscr{F}^{2\mathcal{D}}_{A(\Sigma),X}$ and makes $\varrho \in \mathscr{F}^{\mathrm{EDHML}}_{\Sigma,S}$ hold afterwards;
- $\neg\varrho$ — in the current configuration $\varrho \in \mathscr{F}^{\mathrm{EDHML}}_{\Sigma,S}$ does not hold;
- $\varrho_1 \vee \varrho_2$ — in the current configuration $\varrho_1 \in \mathscr{F}^{\mathrm{EDHML}}_{\Sigma,S}$ or $\varrho_2 \in \mathscr{F}^{\mathrm{EDHML}}_{\Sigma,S}$ hold.

We write $(@s)\varrho$ for $(@^{E(\Sigma)}s)\varrho$, $\Box\varrho$ for $\Box^{E(\Sigma)}\varrho$, $\Diamond^F\varrho$ for $\neg\Box^F\neg\varrho$, $\Diamond\varrho$ for $\Diamond^{E(\Sigma)}\varrho$, $[e(X) /\!\!/ \psi]\varrho$ for $\neg\langle e(X) /\!\!/ \psi \rangle \neg\varrho$, and true for $\downarrow s \,.\, s$.

Example 7.10 An event/data formula can make two kinds of requirements on an event/data structure: On the one hand, it can require the presence of certain mandatory transitions, on the other hand it can require the absence of certain prohibited transitions. Considering the simple UML State Machine in Fig. 7.1, the formula

$$(@s1)\langle\!| \mathsf{inc}(k) : \mathsf{cnt} + k = 4 /\!\!/ \mathsf{cnt}' = 4 |\!\rangle s2$$

requires for each valuation of $\beta : \{k\} \to \mathbb{N}$ such that $\mathsf{cnt} + k = 4$ holds that there is a transition from control state *s1* to control state *s2* for the instantiated event $\mathsf{inc}(\beta)$ where cnt is changed to 4. On the other hand, the formula

$$(@s2)[\mathsf{reset} /\!\!/ \neg(\mathsf{cnt}' = 0)]\mathrm{false}$$

prohibits any transitions out of $s2$ that are labelled with the event reset but do not satisfy $\mathsf{cnt}' = 0$.

In the context of Fig. 7.1, these formulæ only have their explained intended meaning when $s1$ and $s2$ indeed refer to the eponymous states. However, EDHML does not show nominals for explicitly naming control states as part of the State Machine's interface and the reference to specific states always has to build these states' context first using the modalities and the bind operator. On the other hand, as indicated in Sect. 7.2.4, the inclusion of nominals may interfere disadvantageously with the reduct formation.

Let $\sigma : \Sigma \to \Sigma'$ be an event/data signature morphism. The *event/data formulæ translation* $\mathscr{F}^{\text{EDHML}}_{\sigma,S} : \mathscr{F}^{\text{EDHML}}_{\Sigma,S} \to \mathscr{F}^{\text{EDHML}}_{\Sigma',S}$ along σ is recursively given by

- $\mathscr{F}^{\text{EDHML}}_{\sigma,S}(\varphi) = \mathscr{F}^{\mathcal{D}}_{A(\sigma),\emptyset}(\varphi)$;
- $\mathscr{F}^{\text{EDHML}}_{\sigma,S}(s) = s$;
- $\mathscr{F}^{\text{EDHML}}_{\sigma,S}(\downarrow s\,.\,\varrho) = \downarrow s\,.\,\mathscr{F}^{\text{EDHML}}_{\sigma,S\uplus\{s\}}(\varrho)$;
- $\mathscr{F}^{\text{EDHML}}_{\sigma,S}((@^F s)\varrho) = (@^{E(\sigma)(F)} s)\mathscr{F}^{\text{EDHML}}_{\sigma,S}(\varrho)$;
- $\mathscr{F}^{\text{EDHML}}_{\sigma,S}(\Box^F \varrho) = \Box^{E(\sigma)(F)} \mathscr{F}^{\text{EDHML}}_{\sigma,S}(\varrho)$;
- $\mathscr{F}^{\text{EDHML}}_{\sigma,S}(\langle e(X) /\!\!/ \psi\rangle \varrho) = \langle E(\sigma)(e)(X) /\!\!/ \mathscr{F}^{2\mathcal{D}}_{A(\sigma),X}(\psi)\rangle \mathscr{F}^{\text{EDHML}}_{\sigma,S}(\varrho)$;
- $\mathscr{F}^{\text{EDHML}}_{\sigma,S}(\langle\!|e(X) : \phi /\!\!/ \psi|\!\rangle \varrho)$
  $= \langle\!|E(\sigma)(e)(X) : \mathscr{F}^{\mathcal{D}}_{A(\sigma),X}(\phi) /\!\!/ \mathscr{F}^{2\mathcal{D}}_{A(\sigma),X}(\psi)|\!\rangle \mathscr{F}^{\text{EDHML}}_{\sigma,S}(\varrho)$;
- $\mathscr{F}^{\text{EDHML}}_{\sigma,S}(\neg\varrho) = \neg \mathscr{F}^{\text{EDHML}}_{\sigma,S}(\varrho)$;
- $\mathscr{F}^{\text{EDHML}}_{\sigma,S}(\varrho_1 \vee \varrho_2) = \mathscr{F}^{\text{EDHML}}_{\sigma,S}(\varrho_1) \vee \mathscr{F}^{\text{EDHML}}_{\sigma,S}(\varrho_2)$.

The set $\text{Sen}^{\text{EDHML}}(\Sigma)$ of Σ-*event/data sentences* is given by $\mathscr{F}^{\text{EDHML}}_{\Sigma,\emptyset}$, the *event/data sentence translation* $\text{Sen}^{\text{EDHML}}(\sigma) : \text{Sen}^{\text{EDHML}}(\Sigma) \to \text{Sen}^{\text{EDHML}}(\Sigma')$ by $\mathscr{F}^{\text{EDHML}}_{\sigma,\emptyset}$.

7.3.2 Satisfaction Relation for EDHML

The EDHML-satisfaction relation connects EDHML-structures and EDHML-formulæ, expressing whether in some configuration of the structure a particular formula holds with respect to an assignment of control states to state variables. Let Σ be an event/data signature, M a Σ-event/data structure, S a set of state variables, $v : S \to C(M)$ a state variable assignment, and $\gamma \in \Gamma(M)$. The *satisfaction relation* for event/data formulæ is inductively given by

- $M, v, \gamma \models^{\text{EDHML}}_{\Sigma,S} \varphi$ iff $\omega(M)(\gamma) \models^{\mathcal{D}}_{A(\Sigma)} \varphi$;
- $M, v, \gamma \models^{\text{EDHML}}_{\Sigma,S} s$ iff $v(s) = c(M)(\gamma)$;
- $M, v, \gamma \models^{\text{EDHML}}_{\Sigma,S} \downarrow s\,.\,\varrho$ iff $M, v\{s \mapsto c(M)(\gamma)\}, \gamma \models^{\text{EDHML}}_{\Sigma,S\uplus\{s\}} \varrho$;

- $M, v, \gamma \models^{\mathrm{EDHML}}_{\Sigma,S} (@^F s)\varrho$ iff $M, v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma,S} \varrho$
  for all $\gamma' \in \Gamma^F(M)$ with $c(M)(\gamma') = v(s)$;
- $M, v, \gamma \models^{\mathrm{EDHML}}_{\Sigma,S} \Box^F \varrho$ iff $M, v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma,S} \varrho$ for all $\gamma' \in \Gamma^F(M, \gamma)$;
- $M, v, \gamma \models^{\mathrm{EDHML}}_{\Sigma,S} \langle e(X) /\!\!/ \psi \rangle \varrho$ iff there is a $\beta : X \to \mathcal{D}$ and a $\gamma' \in \Gamma(M)$ such that $(\gamma, \gamma') \in R(M)_{e(\beta)}$, $(\omega(M)(\gamma), \omega(M)(\gamma')), \beta \models^{2\mathcal{D}}_{A(\Sigma),X} \psi$, and $M, v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma,S} \varrho$;
- $M, v, \gamma \models^{\mathrm{EDHML}}_{\Sigma,S} \langle\!| e(X) : \phi /\!\!/ \psi |\!\rangle \varrho$
  iff for all $\beta : X \to \mathcal{D}$ with $\omega(M)(\gamma), \beta \models^{\mathcal{D}}_{A(\Sigma),X} \phi$
  there is some $\gamma' \in \Gamma(M)$
  such that $(\gamma, \gamma') \in R(M)_{e(\beta)}$,
  $(\omega(M)(\gamma), \omega(M)(\gamma')), \beta \models^{2\mathcal{D}}_{A(\Sigma),X} \psi$,
  and $M, v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma,S} \varrho$;
- $M, v, \gamma \models^{\mathrm{EDHML}}_{\Sigma,S} \neg\varrho$ iff $M, v, \gamma \not\models^{\mathrm{EDHML}}_{\Sigma,S} \varrho$;
- $M, v, \gamma \models^{\mathrm{EDHML}}_{\Sigma,S} \varrho_1 \vee \varrho_2$ iff $M, v, \gamma \models^{\mathrm{EDHML}}_{\Sigma,S} \varrho_1$ or $M, v, \gamma \models^{\mathrm{EDHML}}_{\Sigma,S} \varrho_2$.

This satisfaction relation is well-behaved with respect to reducts of EDHML-structures. On the one hand, this is due to the use of abstract data names rather than data states in the structures, and on the other hand, to the satisfaction condition of $\mathcal{D}$ and $2\mathcal{D}$.

Lemma 7.11 Let $\sigma : \Sigma \to \Sigma'$ be an event/data signature morphism and M' a Σ'-event/data structure. For all $\varrho \in \mathscr{F}^{\mathrm{EDHML}}_{\Sigma,S}$, all $\gamma' \in \Gamma(M'|\sigma) \subseteq \Gamma(M')$, and all $v : S \to C(M'|\sigma) \subseteq C(M')$ it holds that

$$M'|\sigma, v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma,S} \varrho \iff M', v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma',S} \mathscr{F}^{\mathrm{EDHML}}_{\sigma,S}(\varrho) \,.$$

Proof. We apply induction on the structure of Σ-event/data formulæ.

Case φ:

$$\begin{aligned}
& M'|\sigma, v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma,S} \varphi \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\mathrm{EDHML}} \} \\
& \omega(M'|\sigma)(\gamma') \models^{\mathcal{D}}_{A(\Sigma)} \varphi \\
\Leftrightarrow \quad & \{ \text{ def. } |\sigma \} \\
& \omega(M')(\gamma')|\sigma \models^{\mathcal{D}}_{A(\Sigma)} \varphi \\
\Leftrightarrow \quad & \{ \text{ sat. cond. } \mathcal{D} \} \\
& \omega(M')(\gamma') \models^{\mathcal{D}}_{A(\Sigma')} A(\sigma)(\varphi) \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\mathrm{EDHML}} \} \\
& M', v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma',S} A(\sigma)(\varphi) \\
\Leftrightarrow \quad & \{ \text{ def. } \mathscr{F}^{\mathrm{EDHML}}_{\sigma,S} \} \\
& M', v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma',S} \mathscr{F}^{\mathrm{EDHML}}_{\sigma,S}(\varphi)
\end{aligned}$$

Case s:

$$
\begin{aligned}
& M'|\sigma, v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma,S} s \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\mathrm{EDHML}} \} \\
& v(s) = c(M'|\sigma)(\gamma') \\
\Leftrightarrow \quad & \{ \text{ def. } |\sigma \} \\
& v(s) = c(M')(\gamma') \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\mathrm{EDHML}} \} \\
& M', v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma',S} s \\
\Leftrightarrow \quad & \{ \text{ def. } \mathscr{F}^{\mathrm{EDHML}}_{\sigma,S} \} \\
& M', v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma',S} \mathscr{F}^{\mathrm{EDHML}}_{\sigma,S}(s)
\end{aligned}
$$

Case $\downarrow s \,.\, \varrho$:

$$
\begin{aligned}
& M'|\sigma, v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma,S} \downarrow s \,.\, \varrho \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\mathrm{EDHML}} \} \\
& M'|\sigma, v\{s \mapsto c(M'|\sigma)(\gamma')\}, \gamma' \models^{\mathrm{EDHML}}_{\Sigma, S \uplus \{s\}} \varrho \\
\Leftrightarrow \quad & \{ \text{ def. } |\sigma \text{ and I. H. } \} \\
& M', v\{s \mapsto c(M')(\gamma')\}, \gamma' \models^{\mathrm{EDHML}}_{\Sigma', S \uplus \{s\}} \mathscr{F}^{\mathrm{EDHML}}_{\sigma, S \uplus \{s\}}(\varrho) \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\mathrm{EDHML}} \} \\
& M', v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma', S} \downarrow s \,.\, \mathscr{F}^{\mathrm{EDHML}}_{\sigma, S \uplus \{s\}}(\varrho) \\
\Leftrightarrow \quad & \{ \text{ def. } \mathscr{F}^{\mathrm{EDHML}}_{\sigma,S} \} \\
& M', v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma', S} \mathscr{F}^{\mathrm{EDHML}}_{\sigma,S}(\downarrow s \,.\, \varrho)
\end{aligned}
$$

Case $(@^F s)\varrho$:

$$
\begin{aligned}
& M'|\sigma, v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma,S} (@^F s)\varrho \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\mathrm{EDHML}} \} \\
& M'|\sigma, v, \gamma'' \models^{\mathrm{EDHML}}_{\Sigma,S} \varrho \\
& \quad \text{for all } \gamma'' \in \Gamma^F(M'|\sigma) \text{ with } c(M'|\sigma)(\gamma'') = v(s) \\
\Leftrightarrow \quad & \{ \text{ Lem. 7.8(3) } \} \\
& M'|\sigma, v, \gamma'' \models^{\mathrm{EDHML}}_{\Sigma,S} \varrho \\
& \quad \text{for all } \gamma'' \in \Gamma^{E(\sigma)(F)}(M') \text{ with } c(M'|\sigma)(\gamma'') = v(s) \\
\Leftrightarrow \quad & \{ \text{ def. } |\sigma \text{ and I. H. } \} \\
& M', v, \gamma'' \models^{\mathrm{EDHML}}_{\Sigma',S} \mathscr{F}^{\mathrm{EDHML}}_{\sigma,S}(\varrho) \\
& \quad \text{for all } \gamma'' \in \Gamma^{E(\sigma)(F)}(M') \text{ with } c(M')(\gamma'') = v(s) \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\mathrm{EDHML}} \} \\
& M', v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma',S} (@^{E(\sigma)(F)} s)\mathscr{F}^{\mathrm{EDHML}}_{\sigma,S}(\varrho) \\
\Leftrightarrow \quad & \{ \text{ def. } \mathscr{F}^{\mathrm{EDHML}}_{\sigma,S} \}
\end{aligned}
$$

$$M', v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma', S} \mathscr{F}^{\mathrm{EDHML}}_{\sigma, S}((@^F s)\varrho)$$

Case $\Box^F \varrho$:

$$\begin{array}{ll}
& M'|\sigma, v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma, S} \Box^F \varrho \\
\Leftrightarrow & \{\ \text{def. } \models^{\mathrm{EDHML}}\ \} \\
& M'|\sigma, v, \gamma'' \models^{\mathrm{EDHML}}_{\Sigma, S} \varrho \quad \text{for all } \gamma'' \in \Gamma^F(M'|\sigma, \gamma') \\
\Leftrightarrow & \{\ \text{Lem. 7.8(2)}\ \} \\
& M'|\sigma, v, \gamma'' \models^{\mathrm{EDHML}}_{\Sigma, S} \varrho \quad \text{for all } \gamma'' \in \Gamma^{E(\sigma)(F)}(M', \gamma') \\
\Leftrightarrow & \{\ \text{I.H.}\ \} \\
& M', v, \gamma'' \models^{\mathrm{EDHML}}_{\Sigma', S} \mathscr{F}^{\mathrm{EDHML}}_{\sigma, S}(\varrho) \quad \text{for all } \gamma'' \in \Gamma^{E(\sigma)(F)}(M', \gamma') \\
\Leftrightarrow & \{\ \text{def. } \models^{\mathrm{EDHML}}\ \} \\
& M', v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma', S} \Box^{E(\sigma)(F)} \mathscr{F}^{\mathrm{EDHML}}_{\sigma, S}(\varrho) \\
\Leftrightarrow & \{\ \text{def. } \mathscr{F}^{\mathrm{EDHML}}_{\sigma, S}\ \} \\
& M', v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma', S} \mathscr{F}^{\mathrm{EDHML}}_{\sigma, S}(\Box^F \varrho)
\end{array}$$

Case $\langle e(X) /\!\!/ \psi \rangle \varrho$:

$$\begin{array}{ll}
& M'|\sigma, v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma, S} \langle e(X) /\!\!/ \psi \rangle \varrho \\
\Leftrightarrow & \{\ \text{def. } \models^{\mathrm{EDHML}}\ \} \\
& M'|\sigma, v, \gamma'' \models^{\mathrm{EDHML}}_{\Sigma, S} \varrho \quad \text{for some } \beta : X \to \mathcal{D},\ \gamma'' \in \Gamma(M'|\sigma) \text{ with} \\
& \qquad (\gamma', \gamma'') \in R(M'|\sigma)_{e(\beta)} \text{ and} \\
& \qquad (\omega(M'|\sigma)(\gamma'), \omega(M'|\sigma)(\gamma'')), \beta \models^{2\mathcal{D}}_{A(\Sigma), X} \psi \\
\Leftrightarrow & \{\ \text{Lem. 7.8(1)}\ \} \\
& M'|\sigma, v, \gamma'' \models^{\mathrm{EDHML}}_{\Sigma, S} \varrho \quad \text{for some } \beta : X \to \mathcal{D},\ \gamma'' \in \Gamma(M') \text{ with} \\
& \qquad (\gamma', \gamma'') \in R(M')_{E(\sigma)(e)(\beta)} \text{ and} \\
& \qquad (\omega(M'|\sigma)(\gamma'), \omega(M'|\sigma)(\gamma'')), \beta \models^{2\mathcal{D}}_{A(\Sigma), X} \psi \\
\Leftrightarrow & \{\ \text{def. } |\sigma\ \} \\
& M'|\sigma, v, \gamma'' \models^{\mathrm{EDHML}}_{\Sigma, S} \varrho \quad \text{for some } \beta : X \to \mathcal{D},\ \gamma'' \in \Gamma(M') \text{ with} \\
& \qquad (\gamma', \gamma'') \in R(M')_{E(\sigma)(e)(\beta)} \text{ and} \\
& \qquad (\omega(M')(\gamma')|\sigma, \omega(M')(\gamma'')|\sigma), \beta \models^{2\mathcal{D}}_{A(\Sigma), X} \psi \\
\Leftrightarrow & \{\ \text{sat. cond. } 2\mathcal{D}\ \} \\
& M'|\sigma, v, \gamma'' \models^{\mathrm{EDHML}}_{\Sigma, S} \varrho \quad \text{for some } \beta : X \to \mathcal{D},\ \gamma'' \in \Gamma(M') \text{ with} \\
& \qquad (\gamma', \gamma'') \in R(M')_{E(\sigma)(e)(\beta)} \text{ and} \\
& \qquad (\omega(M')(\gamma'), \omega(M')(\gamma'')), \beta \models^{2\mathcal{D}}_{A(\Sigma'), X} \mathscr{F}^{2\mathcal{D}}_{A(\sigma), X}(\psi) \\
\Leftrightarrow & \{\ \text{I.H.}\ \}
\end{array}$$

$$M', v, \gamma'' \models^{\mathrm{EDHML}}_{\Sigma',S} \mathscr{F}^{\mathrm{EDHML}}_{\sigma,S}(\varrho) \quad \text{for some } \beta : X \to \mathcal{D},\ \gamma'' \in \Gamma(M') \text{ with } (\gamma', \gamma'') \in R(M')_{E(\sigma)(e)(\beta)} \text{ and } (\omega(M')(\gamma'), \omega(M')(\gamma'')), \beta \models^{2\mathcal{D}}_{A(\Sigma'),X} \mathscr{F}^{2\mathcal{D}}_{A(\sigma),X}(\psi)$$

$\Leftrightarrow$ $\{$ def. $\models^{\mathrm{EDHML}}$ $\}$

$$M', v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma',S} \langle E(\sigma)(e)(X) /\!\!/ \mathscr{F}^{2\mathcal{D}}_{A(\sigma),X}(\psi) \rangle \mathscr{F}^{\mathrm{EDHML}}_{\sigma,S}(\varrho)$$

$\Leftrightarrow$ $\{$ def. $\mathscr{F}^{\mathrm{EDHML}}_{\sigma,S}$ $\}$

$$M', v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma',S} \mathscr{F}^{\mathrm{EDHML}}_{\sigma,S}(\langle e(X) /\!\!/ \psi \rangle \varrho)$$

Case $\langle\!| e(X) : \phi /\!\!/ \psi |\!\rangle \varrho$:

$$M'|\sigma, v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma,S} \langle\!| e(X) : \phi /\!\!/ \psi |\!\rangle \varrho$$

$\Leftrightarrow$ $\{$ def. $\models^{\mathrm{EDHML}}$ $\}$

$$M'|\sigma, v, \gamma'' \models^{\mathrm{EDHML}}_{\Sigma,S} \varrho$$
for all $\beta : X \to \mathcal{D}$ such that $\omega(M'|\sigma)(\gamma'), \beta \models^{\mathcal{D}}_{A(\Sigma),X} \phi$ and some $\gamma'' \in \Gamma(M'|\sigma)$ with $(\gamma', \gamma'') \in R(M'|\sigma)_{e(\beta)}$ and $(\omega(M'|\sigma)(\gamma'), \omega(M'|\sigma)(\gamma'')), \beta \models^{2\mathcal{D}}_{A(\Sigma),X} \psi$

$\Leftrightarrow$ $\{$ Lem. 7.8(1) $\}$

$$M'|\sigma, v, \gamma'' \models^{\mathrm{EDHML}}_{\Sigma,S} \varrho$$
for all $\beta : X \to \mathcal{D}$ such that $\omega(M'|\sigma)(\gamma'), \beta \models^{\mathcal{D}}_{A(\Sigma),X} \phi$ and some $\gamma'' \in \Gamma(M')$ with $(\gamma', \gamma'') \in R(M')_{E(\sigma)(e)(\beta)}$ and $(\omega(M'|\sigma)(\gamma'), \omega(M'|\sigma)(\gamma'')), \beta \models^{2\mathcal{D}}_{A(\Sigma),X} \psi$

$\Leftrightarrow$ $\{$ def. $|\sigma$ $\}$

$$M'|\sigma, v, \gamma'' \models^{\mathrm{EDHML}}_{\Sigma,S} \varrho$$
for all $\beta : X \to \mathcal{D}$ such that $\omega(M')(\gamma')|\sigma, \beta \models^{\mathcal{D}}_{A(\Sigma),X} \phi$ and some $\gamma'' \in \Gamma(M')$ with $(\gamma', \gamma'') \in R(M')_{E(\sigma)(e)(\beta)}$ and $(\omega(M')(\gamma')|\sigma, \omega(M')(\gamma'')|\sigma), \beta \models^{2\mathcal{D}}_{A(\Sigma),X} \psi$

$\Leftrightarrow$ $\{$ sat. cond. $\mathcal{D}$, sat. cond. $2\mathcal{D}$ $\}$

$$M'|\sigma, v, \gamma'' \models^{\mathrm{EDHML}}_{\Sigma,S} \varrho$$
for all $\beta : X \to \mathcal{D}$ such that $\omega(M')(\gamma'), \beta \models^{\mathcal{D}}_{A(\Sigma'),X} \mathscr{F}^{\mathcal{D}}_{A(\sigma),X}(\phi)$ and some $\gamma'' \in \Gamma(M')$ with $(\gamma', \gamma'') \in R(M')_{E(\sigma)(e)(\beta)}$ and $(\omega(M')(\gamma'), \omega(M')(\gamma'')), \beta \models^{2\mathcal{D}}_{A(\Sigma'),X} \mathscr{F}^{2\mathcal{D}}_{A(\sigma),X}(\psi)$

$\Leftrightarrow$ $\{$ I. H. $\}$

$$M', v, \gamma'' \models^{\mathrm{EDHML}}_{\Sigma',S} \mathscr{F}^{\mathrm{EDHML}}_{\sigma,S}(\varrho)$$
for all $\beta : X \to \mathcal{D}$ such that $\omega(M')(\gamma'), \beta \models^{\mathcal{D}}_{A(\Sigma'),X} \mathscr{F}^{\mathcal{D}}_{A(\sigma),X}(\phi)$ and some $\gamma'' \in \Gamma(M')$ with $(\gamma', \gamma'') \in R(M')_{E(\sigma)(e)(\beta)}$ and $(\omega(M')(\gamma'), \omega(M')(\gamma'')), \beta \models^{2\mathcal{D}}_{A(\Sigma'),X} \mathscr{F}^{2\mathcal{D}}_{A(\sigma),X}(\psi)$

$\Leftrightarrow$ $\{$ def. $\models^{\mathrm{EDHML}}$ $\}$

$$
\begin{aligned}
& M', v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma', S} \langle\!|E(\sigma)(e)(X) : \mathscr{F}^{\mathcal{D}}_{A(\sigma),X}(\phi) /\!/ \mathscr{F}^{2\mathcal{D}}_{A(\sigma),X}(\psi) |\!\rangle \mathscr{F}^{\mathrm{EDHML}}_{\sigma,S}(\varrho) \\
\Leftrightarrow \quad & \{ \text{ def. } \mathscr{F}^{\mathrm{EDHML}}_{\sigma,S} \} \\
& M', v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma', S} \mathscr{F}^{\mathrm{EDHML}}_{\sigma,S}(\langle\!| e(X) : \phi /\!/ \psi |\!\rangle \varrho)
\end{aligned}
$$

Case $\neg\varrho$:

$$
\begin{aligned}
& M'|\sigma, v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma, S} \neg\varrho \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\mathrm{EDHML}} \} \\
& M'|\sigma, v, \gamma' \not\models^{\mathrm{EDHML}}_{\Sigma, S} \varrho \\
\Leftrightarrow \quad & \{ \text{ I. H. } \} \\
& M', v, \gamma' \not\models^{\mathrm{EDHML}}_{\Sigma', S} \mathscr{F}^{\mathrm{EDHML}}_{\sigma,S}(\varrho) \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\mathrm{EDHML}} \} \\
& M', v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma', S} \neg\mathscr{F}^{\mathrm{EDHML}}_{\sigma,S}(\varrho) \\
\Leftrightarrow \quad & \{ \text{ def. } \mathscr{F}^{\mathrm{EDHML}}_{\sigma,S} \} \\
& M', v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma', S} \mathscr{F}^{\mathrm{EDHML}}_{\sigma,S}(\neg\varrho)
\end{aligned}
$$

Case $\varrho \vee \varsigma$:

$$
\begin{aligned}
& M'|\sigma, v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma, S} \varrho \vee \varsigma \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\mathrm{EDHML}} \} \\
& M'|\sigma, v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma, S} \varrho \text{ or } M'|\sigma, v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma, S} \varsigma \\
\Leftrightarrow \quad & \{ \text{ I. H. } \} \\
& M', v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma', S} \mathscr{F}^{\mathrm{EDHML}}_{\sigma,S}(\varrho) \text{ or } M', v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma', S} \mathscr{F}^{\mathrm{EDHML}}_{\sigma,S}(\varsigma) \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\mathrm{EDHML}} \} \\
& M', v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma', S} \mathscr{F}^{\mathrm{EDHML}}_{\sigma,S}(\varrho) \vee \mathscr{F}^{\mathrm{EDHML}}_{\sigma,S}(\varsigma) \\
\Leftrightarrow \quad & \{ \text{ def. } \mathscr{F}^{\mathrm{EDHML}}_{\sigma,S} \} \\
& M', v, \gamma' \models^{\mathrm{EDHML}}_{\Sigma', S} \mathscr{F}^{\mathrm{EDHML}}_{\sigma,S}(\varrho \vee \varsigma)
\end{aligned}
$$

□

For a $\Sigma \in |\mathrm{Sig}^{\mathrm{EDHML}}|$, an $M \in |\mathrm{Mod}^{\mathrm{EDHML}}(\Sigma)|$, and a $\rho \in \mathrm{Sen}^{\mathrm{EDHML}}(\Sigma)$ the *satisfaction relation* $M \models^{\mathrm{EDHML}}_{\Sigma} \rho$ holds if, and only if, $M, \emptyset, \gamma_0 \models^{\mathrm{EDHML}}_{\Sigma,\emptyset} \rho$ for all $\gamma_0 \in \Gamma_0(M)$.

Theorem 7.12 $(\mathrm{Sig}^{\mathrm{EDHML}}, \mathrm{Mod}^{\mathrm{EDHML}}, \mathrm{Sen}^{\mathrm{EDHML}}, \models^{\mathrm{EDHML}})$ is an institution.

Proof. The satisfaction condition that for any $\sigma : \Sigma \to \Sigma'$ in $\mathrm{Sig}^{\mathrm{EDHML}}$, $M' \in |\mathrm{Mod}^{\mathrm{EDHML}}(\Sigma')|$, and $\rho \in \mathrm{Sen}^{\mathrm{EDHML}}(\Sigma)$, it holds that

$$
\mathrm{Mod}^{\mathrm{EDHML}}(\sigma)(M') \models^{\mathrm{EDHML}}_{\Sigma} \rho \iff M' \models^{\mathrm{EDHML}}_{\Sigma'} \mathrm{Sen}^{\mathrm{EDHML}}(\sigma)(\rho)
$$

directly follows from Lem. 7.11. □

Alg. 1 Constructing an EDHML-sentence from a set of transition specifications

Require: $T \equiv$ a set of transition specifications
$Im_T(c) = \{(\phi, e(X), \psi, c') \mid (c, \phi, e(X), \psi, c') \in T\}$
$Im_T(c, e(X)) = \{(\phi, \psi, c') \mid (c, \varphi, e(X), \psi, c') \in T\}$

function $\mathrm{sen}(c, I, V, B)$ ▷ c: state, I: image to visit, V: states to visit, B: bound states
  if $I \neq \emptyset$ **then**
    $(\phi, e(X), \psi, c') \leftarrow$ **choose** I
    if $c' \in B$ **then**
      return $(@c)\langle\!\langle e(X) : \phi /\!\!/ \psi \rangle\!\rangle (c' \wedge \mathrm{sen}(c, I \setminus \{(\phi, e(X), \psi, c')\}, V, B))$
    else
      return $(@c)\langle\!\langle e(X) : \phi /\!\!/ \psi \rangle\!\rangle (\downarrow c' . \mathrm{sen}(c, I \setminus \{(\phi, e(X), \psi, c')\}, V, B \cup \{c'\}))$
  $V \leftarrow V \setminus \{c\}$
  if $V \neq \emptyset$ **then**
    $c' \leftarrow$ **choose** $B \cap V$
    return $\mathrm{sen}(c', Im_T(c'), V, B)$
  return $(\bigwedge_{c \in B} \mathrm{fin}(c)) \wedge \bigwedge_{c_1 \in B, c_2 \in B \setminus \{c_1\}} \neg(@c_1)c_2$

function $\mathrm{fin}(c)$
  return $(@c) \bigwedge_{e(X) \in E(\Sigma(U))} \bigwedge_{P \subseteq Im_T(c, e(X))}$
    $[e(X) /\!\!/ \big(\bigwedge_{(\phi,\psi,c') \in P} (\phi \wedge \psi) \big) \wedge$
      $\neg \big(\bigvee_{(\phi,\psi,c') \in Im_T(c,e(X)) \setminus P} (\phi \wedge \psi) \big)] \big(\bigvee_{(\phi,\psi,c') \in P} c' \big)$

7.3.3 Representing Simple UML State Machines in EDHML

The hybrid modal logic EDHML is expressive enough to characterise the model class of a simple UML State Machine U by a single sentence ϱ_U, i.e., an event/data structure M is a model of U if, and only if, $M \models^{\mathrm{EDHML}}_{\Sigma(U)} \varrho_U$. Such a characterisation is achieved by means of Alg. 1 that is a slight variation of the characterisation algorithm for so-called operational specifications within $\mathcal{E}^{\downarrow}$ [HMK19] by including also events with data arguments. The algorithm constructs a sentence expressing that

– all semantic transitions according to the explicit syntactic transition specifications are indeed possible and that

– no further semantic transitions exist, i.e., every semantic transition adheres to some syntactic transition.

This sentence is expected to hold in the initial configuration and characterizes the entire machine. State names from the input are reused as variable names for control states. All state variables except the initial state are bound. Their distinctness is ensured (see the $\neg(@c_1)c_2$ in Alg. 1). Together, this ensures state names can be consistently renamed, but are otherwise fixed.

The algorithm proceeds as follows: For a set of transition specifications T, a call $\mathrm{sen}(c, I, V, B)$ performs a recursive breadth-first traversal[1] starting from c, where I holds the unprocessed

[1] We are traversing the graph spanned by the transition specifications, i.e., by the elements of T.

quadruples[2] $(\phi, e(X), \psi, c')$ of transitions in T outgoing from c, V the remaining states to visit, and B the set of already bound states. The function first requires the existence of each outgoing transition of I in the resulting formula, binding any newly reached state. Having visited all states in V, it requires that no other transitions from the states in B exist using calls to fin, and adds the requirement that all states in B are pairwise different. Formula $\mathrm{fin}(c)$ expresses that at c, for all events $e(X)$ and for all subsets P of the transitions in T outgoing from c, whenever an $e(X)$-transition can be done with the combined effect of P but not adhering to any of the effects of the transitions currently not selected, the $e(X)$-transition must have one of the states as its target that are target states of P.

Example 7.13 Applying Alg. 1 to the set of explicitly mentioned, "red" transition specifications T of the simple UML State Machine *Counter* in Fig. 7.1, i.e., calling $\mathrm{sen}(s1, Im_T(s1), \{s1, s2\}, \{s1\})$ yields $\varrho_{s1,s1}$ with

$$
\begin{aligned}
\varrho_{s1,s1} &= (@s1)\langle\!|\mathsf{inc}(k) : \mathsf{cnt} + k \leq 4 /\!\!/ \mathsf{cnt}' = \mathsf{cnt} + k|\!\rangle (s1 \wedge \varrho_{s1,s2}) \\
\varrho_{s1,s2} &= (@s1)\langle\!|\mathsf{inc}(k) : \mathsf{cnt} + k = 4 /\!\!/ \mathsf{cnt}' = \mathsf{cnt} + k|\!\rangle \downarrow s2\,.\,(\varrho_{s2,s1}) \\
\varrho_{s2,s1} &= (@s2)\langle\!|\mathsf{reset} : \mathsf{true} /\!\!/ \mathsf{cnt}' = 0|\!\rangle (s1 \wedge \varrho_{\mathrm{fin}}) \\
\varrho_{\mathrm{fin}} &= \varrho_{\mathrm{fin}(s1)} \wedge \varrho_{\mathrm{fin}(s2)} \wedge \neg(@s1)s2 \\
\varrho_{\mathrm{fin}(s1)} &= (@s1)\big([\mathsf{inc}(k) /\!\!/ \neg((\mathsf{cnt} + k \leq 4 \wedge \mathsf{cnt}' = \mathsf{cnt} + k) \vee \\
&\qquad (\mathsf{cnt} + k = 4 \wedge \mathsf{cnt}' = 4))]\mathsf{false} \wedge \\
&\qquad [\mathsf{inc}(k) /\!\!/ (\mathsf{cnt} + k \leq 4 \wedge \mathsf{cnt}' = \mathsf{cnt} + k) \wedge \\
&\qquad\quad \neg(\mathsf{cnt} + k = 4 \wedge \mathsf{cnt}' = 4)]s1 \wedge \\
&\qquad [\mathsf{inc}(k) /\!\!/ (\mathsf{cnt} + k = 4 \wedge \mathsf{cnt}' = 4) \wedge \\
&\qquad\quad \neg(\mathsf{cnt} + k \leq 4 \wedge \mathsf{cnt}' = \mathsf{cnt} + k)]s2 \wedge \\
&\qquad [\mathsf{inc}(k) /\!\!/ (\mathsf{cnt} + k \leq 4 \wedge \mathsf{cnt}' = \mathsf{cnt} + k) \wedge \\
&\qquad\quad (\mathsf{cnt} + k = 4 \wedge \mathsf{cnt}' = 4)](s1 \vee s2) \wedge \\
&\qquad [\mathsf{reset} /\!\!/ \mathsf{true}]\mathsf{false}\big) \\
\varrho_{\mathrm{fin}(s2)} &= (@s2)\big([\mathsf{inc}(k) /\!\!/ \mathsf{true}]\mathsf{false} \wedge \\
&\qquad [\mathsf{reset} /\!\!/ \neg(\mathsf{cnt}' = 0)]\mathsf{false} \wedge \\
&\qquad [\mathsf{reset} /\!\!/ \mathsf{cnt}' = 0]s1\big)
\end{aligned}
$$

In fact, there is no outgoing "red" transition for reset from $s1$, thus $P = \emptyset$ is the only choice for this event in $\mathrm{fin}(s1)$ and the clause $[\mathsf{reset} /\!\!/ \mathsf{true}]\mathsf{false}$ is included. For $\mathsf{inc}(k)$ there are two outgoing transitions resulting in four different clauses checking whether none, the one or the other, or both transitions are executable.

In order to apply the algorithm to simple UML State Machines, the idling self-loops for achieving input-enabledness first have to be made explicit. For a syntactically input-enabled simple UML State Machine U a characterising sentence then reads

$$\varrho_U = \downarrow c_0\,.\,\varphi_0 \wedge \mathrm{sen}(c_0, Im_{T(U)}(c_0), C(U), \{c_0\})\ ,$$

[2] To recall from Sect. 7.2.2: A transition specification consists of an start control state c, a (data) state formula ϕ acting as a guard, an event $e(X)$ acting as a trigger, a (data) transition formula ψ encoding the assignment actions, and resulting control state c'.

where $c_0 = c_0(U)$ and $\varphi_0 = \varphi_0(U)$. Due to syntactic reachability, the bound states B of Alg. 1 become $C(U)$ when sen is called for $B = \{c_0(U)\}$ and V reaches $\emptyset$.

Chapter 8

A Theoroidal Comorphism from EDHML to CASL

Contents

In this chapter, we define a theoroidal comorphism from EDHML to CASL[1]. The construction mainly follows the standard translation of modal logics to first-order logic [VPMY04] which has been considered for hybrid logics also on an institutional level [Mad13, DM16].

The contents of this chapter have appeared in similar form as part of our paper at WADT 2020 [RBKR20].

The construction of the comorphism was a common effort between Alexander Knapp, Markus Roggenbach and myself. In our virtual meetings, we jointly sketched the constructions, proofs and examples, which I then worked out in detail and wrote down in their final form.

8.1 Comorphism Definition

The basis is a representation of EDHML-signatures and the frame given by EDHML-structures as a CASL-specification as shown in Fig. 8.1. The signature translation

$$\nu^{\mathrm{Sig}} : \mathrm{Sig}^{\mathrm{EDHML}} \to \mathrm{Pres}^{\mathrm{CASL}}$$

maps an EDHML-signature Σ to the CASL-theory presentation given by TRANS_Σ and a EDHML-signature morphism to the corresponding theory presentation morphism.

We will now give a line-by-line discussion of the translation frame in Fig. 8.1 for TRANS_Σ.

[1] The translation indeed lands in CASL. However, as one of the referees pointed out, the *structured free* construct, which we use to define an inductive predicate, is not part of the CASL institution, but rather of CASL's institution independent structuring constructs. Structured specifactions over an arbitrary institution again form institutions, for many notions of structuring [DM03]. In particular, this is the case for freeness constraints [GB92, Def. 25, Prop. 26]. Our comorphism technically lands in such an institution over CASL.

from *Basic/StructuredDatatypes* **get** SET % import finite sets
spec $\text{TRANS}_\Sigma = Dt$
then free type $\text{Evt} ::= \tau_\text{e}(E(\Sigma))$
$\quad\quad$ % $\tau_\text{e}(\{e(X)\}) = e(dt^{|X|})$, $\tau_\text{e}(\{e(X)\} \cup E) = e(dt^{|X|}) \mid \tau_\text{e}(E)$
$\quad$ **free type** $\text{EvtNm} ::= \tau_\text{n}(E(\Sigma))$ % $\tau_\text{n}(\{e(X)\}) = e$, $\tau_\text{n}(\{e(X)\} \cup E) = e \mid \tau_\text{n}(E)$
$\quad$ **op** $\text{nm} : \text{Evt} \to \text{EvtNm}$
$\quad$ **axiom** $\forall x_1, \dots, x_n : dt \bullet \text{nm}(e(x_1, \dots, x_n)) = e$ % for each $e(x_1, \dots, x_n) \in E(\Sigma)$
then SET[sort EvtNm]
then sort Ctrl
$\quad$ **free type** $\text{Conf} ::= \text{conf}(\text{c} : \text{Ctrl}; \tau_\text{a}(A(\Sigma)))$
$\quad\quad$ % $\tau_\text{a}(\{a\}) = a : dt$, $\tau_\text{a}(\{a\} \cup A) = a : dt; \tau_\text{a}(A)$
$\quad$ **preds** $\text{init} : \text{Conf}$;
$\quad\quad$ $\text{trans} : \text{Conf} \times \text{Evt} \times \text{Conf}$
$\quad$ $\bullet\ \exists g : \text{Conf} \bullet \text{init}(g)$ % there is some initial configuration
$\quad$ $\bullet\ \forall g, g' : \text{Conf} \bullet \text{init}(g) \wedge \text{init}(g') \Rightarrow \text{c}(g) = \text{c}(g')$ % single initial control state
$\quad$ **free** { **pred** $\text{reachable} : \mathit{Set}[\text{EvtNm}] \times \text{Conf} \times \text{Conf}$
$\quad\quad$ $\forall g, g', g'' : \text{Conf}, E : \mathit{Set}[\text{EvtNm}], e : \text{Evt}$
$\quad\quad$ $\bullet\ \text{reachable}(E, g, g)$
$\quad\quad$ $\bullet\ \text{reachable}(E, g, g') \wedge \text{nm}(e) \in E \wedge \text{trans}(g', e, g'') \Rightarrow \text{reachable}(E, g, g'')$ }
$\quad$ **then preds** $\text{reachable}(E : \mathit{Set}[\text{EvtName}], g : \text{Conf}) \Leftrightarrow$
$\quad\quad\quad$ $\exists g_0 : \text{Conf} \bullet \text{init}(g_0) \wedge \text{reachable}(E, g_0, g)$;
$\quad\quad$ $\text{reachable}(g : \text{Conf}) \Leftrightarrow \text{reachable}(E(\Sigma), g)$
end

Figure 8.1: Frame for translating EDHML into CASL

- TRANS_Σ uses finite sets from the CASL standard library.
- TRANS_Σ itself starts with a specification *Dt* of the underlying data universe.
- Next, it introduces a free type `Evt` for events. The case distinction between the different event constructors is here generated by a meta-language function τ_e, which recurses over the event set $|E(\Sigma)|$ from the signature. For each event, we generate a constructor. To recall, each event $e(X)$ comes with a finite set $v(e(X))$ of variables, which we here include as arguments to the event constructor.
- Next, we introduce the free type `EvtNm` for event names. The constructors are generated by the recursive function τ_n, which is similar to τ_e, but leaves out the arguments, i.e., an event name is an event without the data.
- Next, we include into TRANS_Σ the specification of finite sets, specialised to event names as elements.
- Next, we introduce a sort `Ctrl` for the control states.
- Next, we introduce a free type for configurations with only one constructor `conf`. A configuration is given by a control state and, for each attribute, one value from the data

universe dt. The data value arguments to `conf` are generated by the function τ_a, which recurses over the attribute set $A(\Sigma)$ from the signature.

– Next, we introduce the predicate `init` which characterises initial configurations.

– Next, we introduce the transition predicate `trans`, which characterises triples of a configuration before the transition, an event and a configuration after the transition.

– Next, we require by an axiom that there is an initial configuration.

– Next, we require by an axiom that any two initial control states are equal.

– Next, we introduce the predicate `reachable` by a structured free block. In our experiments, we had to use different methods to encode the inductive nature of reachability, due to a lack of tools support for CASL's structured free construct. However, in the formal definition, we considered the structured free to be the clearest way of expressing what is going on here. In the structured free block we

  1. introduce the predicate `reachable`, taking a set of event names (as we are considering reachability using only events with these names) and the configurations in which we start and end;
  2. quantify over variables we will need in the following two axioms;
  3. state the base case axioms, that each configuration is reachable from itself; and
  4. state the step axiom, that transitions with allowed triggers preserve reachability.

– Finally, we introduce two abbreviated reachability predicates by first specialising the starting configuration to any allowed initial configuration, then by specialising the allowed set of event names to all event names from the signature.

The model translation

$$\nu^{\mathrm{Mod}}_{\Sigma} : \mathrm{Mod}^{\mathrm{CASL}}(\nu^{\mathrm{Sig}}(\Sigma)) \to \mathrm{Mod}^{\mathrm{EDHML}}(\Sigma)$$

can rely on the encoding just described. In particular, for a model $M' \in \mathrm{Mod}^{\mathrm{CASL}}(\nu^{\mathrm{Sig}}(\Sigma))$, there are, using the bijection $\iota_{M',dt} : dt^{M'} \cong \mathcal{D}$, an injective map $\iota_{M',\mathrm{Conf}} : \mathrm{Conf}^{M'} \rightarrowtail \mathrm{Ctrl}^{M'} \times \Omega(A(\Sigma))$ and a bijective map $\iota_{M',\mathrm{Evt}} : \mathrm{Evt}^{M'} \cong \{e(\beta) \mid e(X) \in E(\Sigma),\ \beta : X \to \mathcal{D}\}$. The EDHML-structure resulting from a CASL-model of TRANS_{Σ} can thus be defined by

– $\Gamma(\nu^{\mathrm{Mod}}_{\Sigma}(M')) = \iota^{-1}_{M',\mathrm{Conf}}(\{g' \in M'_{\mathrm{Conf}} \mid \mathrm{reachable}^{M'}(g')\})$

– $R(\nu^{\mathrm{Mod}}_{\Sigma}(M'))_{e(\beta)} = \{(\gamma,\gamma') \in \Gamma(\nu^{\mathrm{Mod}}_{\Sigma}(M')) \times \Gamma(\nu^{\mathrm{Mod}}_{\Sigma}(M')) \mid \mathrm{trans}^{M'}(\iota_{M',\mathrm{Conf}}(\gamma), \iota^{-1}_{M',\mathrm{Evt}}(e(\beta)), \iota_{M',\mathrm{Conf}}(\gamma'))\})$

– $\Gamma_0(\nu^{\mathrm{Mod}}_{\Sigma}(M')) = \{\gamma \in \Gamma(\nu^{\mathrm{Mod}}_{\Sigma}(M')) \mid \mathrm{init}^{M'}(\iota_{M',\mathrm{Conf}}(\gamma))\})$

– $\omega(\nu^{\mathrm{Mod}}_{\Sigma}(M')) = \{(c,\omega) \in \Gamma(\nu^{\mathrm{Mod}}_{\Sigma}(M')) \mapsto \omega\}$

For EDHML-sentences, we first define a formula translation

$$\nu^{\mathscr{F}}_{\Sigma,S,g} : \mathscr{F}^{\text{EDHML}}_{\Sigma,S} \to \mathscr{F}^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),S\cup\{g\}}$$

which, mimicking the standard translation, takes a variable $g : \text{Conf}$ as a parameter that records the "current configuration" and also uses a set S of state names for the control states. The translation embeds the data state and 2-data state formulæ using the substitution $A(\Sigma)(g) = \{a \mapsto a(g) \mid a \in A(\Sigma)\}$ for replacing the attributes $a \in A(\Sigma)$ by the accessors $a(g)$. The translation of EDHML-formulæ then reads

- $\nu^{\mathscr{F}}_{\Sigma,S,g}(\varphi) = \mathscr{F}^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),A(\Sigma)(g)}(\varphi)$
- $\nu^{\mathscr{F}}_{\Sigma,S,g}(s) = (s = \mathrm{c}(g))$
- $\nu^{\mathscr{F}}_{\Sigma,S,g}(\downarrow s\,.\,\varrho) = \exists s : \text{Ctrl}\,.\, s = \mathrm{c}(g) \wedge \nu^{\mathscr{F}}_{\Sigma,S\uplus\{s\},g}(\varrho)$
- $\nu^{\mathscr{F}}_{\Sigma,S,g}((@^F s)\varrho) = \forall g' : \text{Conf}\,.\,(\mathrm{c}(g') = s \wedge \text{reachable}(F,g')) \Rightarrow \nu^{\mathscr{F}}_{\Sigma,S,g'}(\varrho)$
- $\nu^{\mathscr{F}}_{\Sigma,S,g}(\Box^F\varrho) = \forall g' : \text{Conf}\,.\,\text{reachable}(F,g,g') \Rightarrow \nu^{\mathscr{F}}_{\Sigma,S,g'}(\varrho)$
- $\nu^{\mathscr{F}}_{\Sigma,S,g}(\langle e(X) /\!\!/ \psi\rangle\varrho) = \exists X : dt\,.\,\exists g' : \text{Conf}\,.\,\text{trans}(g,e(X),g') \wedge \mathscr{F}^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),A(\Sigma)(g)\cup A(\Sigma)(g')\cup 1_X}(\psi) \wedge \nu^{\mathscr{F}}_{\Sigma,S,g'}(\varrho)$
- $\nu^{\mathscr{F}}_{\Sigma,S,g}(\langle\!| e(X) : \phi /\!\!/ \psi |\!\rangle\varrho) = \forall X : dt\,.\,\mathscr{F}^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),A(\Sigma)(g)\cup 1_X}(\phi) \Rightarrow \exists g' : \text{Conf}\,.\,\text{trans}(g,e(X),g') \wedge \mathscr{F}^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),A(\Sigma)(g)\cup A(\Sigma)(g')\cup 1_X}(\psi) \wedge \nu^{\mathscr{F}}_{\Sigma,S,g'}(\varrho)$
- $\nu^{\mathscr{F}}_{\Sigma,S,g}(\neg\varrho) = \neg\nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho)$
- $\nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho_1 \vee \varrho_2) = \nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho_1) \vee \nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho_2)$

Example 8.1 The translation of $(@s1)\langle\!|\mathsf{inc}(x) : \mathsf{cnt} + x \leq 4 /\!\!/ \mathsf{cnt}' = \mathsf{cnt} + x|\!\rangle s1$ over the state set $\{s1\}$ and the configuration variable g is

$$\begin{aligned}
& \nu^{\mathscr{F}}_{\Sigma,\{s1\},g}((@s1)\langle\!|\mathsf{inc}(x) : \mathsf{cnt} + x \leq 4 /\!\!/ \mathsf{cnt}' = \mathsf{cnt} + x|\!\rangle s1) \\
= \;& \forall g' : \text{Conf}\,.\,(\mathrm{c}(g') = s1 \wedge \text{reachable}(g')) \Rightarrow \\
& \quad \nu^{\mathscr{F}}_{\Sigma,\{s1\},g'}(\langle\!|\mathsf{inc}(x) : \mathsf{cnt} + x \leq 4 /\!\!/ \mathsf{cnt}' = \mathsf{cnt} + x|\!\rangle s1) \\
= \;& \forall g' : \text{Conf}\,.\,(\mathrm{c}(g') = s1 \wedge \text{reachable}(g')) \Rightarrow \\
& \quad \forall x : dt\,.\,\mathrm{cnt}(g') + x \leq 4 \Rightarrow \\
& \quad\quad \exists g'' : \text{Conf}\,.\,\text{trans}(g',\mathrm{inc}(x),g'') \wedge \\
& \quad\quad\quad \mathrm{cnt}(g'') = \mathrm{cnt}(g') + x \wedge \nu^{\mathscr{F}}_{\Sigma,\{s1\},g''}(s1) \\
= \;& \forall g' : \text{Conf}\,.\,(\mathrm{c}(g') = s1 \wedge \text{reachable}(g')) \Rightarrow \\
& \quad \forall x : dt\,.\,\mathrm{cnt}(g') + x \leq 4 \Rightarrow \\
& \quad\quad \exists g'' : \text{Conf}\,.\,\text{trans}(g',\mathrm{inc}(x),g'') \wedge \\
& \quad\quad\quad \mathrm{cnt}(g'') = \mathrm{cnt}(g') + x \wedge s1 = \mathrm{c}(g'')
\end{aligned}$$

Building on the translation of formulæ, the sentence translation

$$\nu^{\mathrm{Sen}}_{\Sigma} : \mathrm{Sen}^{\mathrm{EDHML}}(\Sigma) \to \mathrm{Sen}^{\textsc{Casl}}(\nu^{\mathrm{Sig}}(\Sigma))$$

only has to require additionally that evaluation starts in an initial state, as sentences are stated from the perspective of an initial state. Thus, sentence translation is defined by:

– $\nu^{\mathrm{Sen}}_{\Sigma}(\rho) = \forall g : \mathrm{Conf}.\mathrm{init}(g) \Rightarrow \nu^{\mathscr{F}}_{\Sigma,\emptyset,g}(\rho)$

8.2 Satisfaction Condition

The translation of CASL-models of $\textsc{Trans}_{\Sigma}$ into EDHML-structures and the translation of EDHML-formulæ into CASL-formulæ over $\textsc{Trans}_{\Sigma}$ fulfil the requirements of the "open" satisfaction condition of theoroidal comorphisms:

Lemma 8.2 For a $\varrho \in \mathscr{F}^{\mathrm{EDHML}}_{\Sigma,S}$, an $M' \in \mathrm{Mod}^{\textsc{Casl}}(\nu^{\mathrm{Sig}}(\Sigma))$, a $v : S \to C(\nu^{\mathrm{Mod}}_{\Sigma}(M'))$, and a $\gamma \in \Gamma(\nu^{\mathrm{Mod}}_{\Sigma}(M'))$ it holds with $\beta'_{M',g}(v,\gamma) = \iota^{-1}_{M',\mathrm{Ctrl}} \circ v \cup \{g \mapsto \iota_{M',\mathrm{Conf}}(\gamma)\}$ that

$$\nu^{\mathrm{Mod}}_{\Sigma}(M'), v, \gamma \models^{\mathrm{EDHML}}_{\Sigma,S} \varrho \iff M', \beta'_{M',g}(v,\gamma) \models^{\textsc{Casl}}_{\nu^{\mathrm{Sig}}(\Sigma),S\cup\{g\}} \nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho)\ .$$

Proof. We apply induction on the structure of Σ-event/data formulæ.

Case φ:

$$\begin{array}{ll}
& \nu^{\mathrm{Mod}}_{\Sigma}(M'), v, \gamma \models^{\mathrm{EDHML}}_{\Sigma,S} \varphi \\
\Leftrightarrow & \{\ \text{def. } \models^{\mathrm{EDHML}}\ \} \\
& \omega(\nu^{\mathrm{Mod}}_{\Sigma}(M'))(\gamma) \models^{\mathcal{D}}_{A(\Sigma)} \varphi \\
\Leftrightarrow & \{\ \text{def. } \models^{\mathcal{D}}, \text{def. } \models^{\textsc{Casl}}\ \} \\
& M', \iota^{-1}_{M',dt} \circ \omega(\nu^{\mathrm{Mod}}_{\Sigma}(M'))(\gamma) \cup \\
& \quad \{g \mapsto \iota_{M',\mathrm{Conf}}(\gamma)\} \models^{\textsc{Casl}}_{\nu^{\mathrm{Sig}}(\Sigma),\{g\}} \mathscr{F}^{\textsc{Casl}}_{\nu^{\mathrm{Sig}}(\Sigma),A(\Sigma)(g)}(\varphi) \\
\Leftrightarrow & \{\ \text{def. } \models^{\textsc{Casl}}\ \} \\
& M', \beta'_{M',g}(v,\gamma) \models^{\textsc{Casl}}_{\nu^{\mathrm{Sig}}(\Sigma),S\cup\{g\}} \mathscr{F}^{\textsc{Casl}}_{\nu^{\mathrm{Sig}}(\Sigma),A(\Sigma)(g)}(\varphi) \\
\Leftrightarrow & \{\ \text{def. } \nu^{\mathscr{F}}\ \} \\
& M', \beta'_{M',g}(v,\gamma) \models^{\textsc{Casl}}_{\nu^{\mathrm{Sig}}(\Sigma),S\cup\{g\}} \nu^{\mathscr{F}}_{\Sigma,S,g}(\varphi)
\end{array}$$

Case s:

$$\begin{array}{ll}
& \nu^{\mathrm{Mod}}_{\Sigma}(M'), v, \gamma \models^{\mathrm{EDHML}}_{\Sigma,S} s \\
\Leftrightarrow & \{\ \text{def. } \models^{\mathrm{EDHML}}\ \} \\
& v(s) = c(\nu^{\mathrm{Mod}}_{\Sigma}(M'))(\gamma) \\
\Leftrightarrow & \{\ \text{def. } \nu^{\mathrm{Mod}}, \models^{\textsc{Casl}}\ \} \\
& M', \beta'_{M',g}(v,\gamma) \models^{\textsc{Casl}}_{\nu^{\mathrm{Sig}}(\Sigma),S\cup\{g\}} s = \mathrm{c}(g) \\
\Leftrightarrow & \{\ \text{def. } \nu^{\mathscr{F}}\ \}
\end{array}$$

$$M', \beta'_{M',g}(v,\gamma) \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),S\cup\{g\}} \nu^{\mathscr{F}}_{\Sigma,S,g}(s)$$

Case $\downarrow s\,.\,\varrho$:

$$\begin{array}{ll}
& \nu^{\text{Mod}}_{\Sigma}(M'), v, \gamma \models^{\text{EDHML}}_{\Sigma,S} \downarrow s\,.\,\varrho \\
\Leftrightarrow & \{\text{ def. } \models^{\text{EDHML}} \} \\
& \nu^{\text{Mod}}_{\Sigma}(M'), v\{s \mapsto c(\nu^{\text{Mod}}_{\Sigma}(M'))(\gamma)\}, \gamma \models^{\text{EDHML}}_{\Sigma,S\uplus\{s\}} \varrho \\
\Leftrightarrow & \{\text{ I.H. } \} \\
& M', \beta'_{M',g}(v\{s \mapsto c(\nu^{\text{Mod}}_{\Sigma}(M'))(\gamma)\}, \gamma) \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),(S\uplus\{s\})\cup\{g\}} \nu^{\mathscr{F}}_{\Sigma,S\uplus\{s\},g}(\varrho) \\
\Leftrightarrow & \{\text{ def. } \models^{\text{CASL}} \} \\
& M', \beta'_{M',g}(v,\gamma) \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),S\cup\{g\}} \exists s : \text{Ctrl}\,.\, s = \text{c}(g) \wedge \nu^{\mathscr{F}}_{\Sigma,S\uplus\{s\},g}(\varrho) \\
\Leftrightarrow & \{\text{ def. } \nu^{\mathscr{F}} \} \\
& M', \beta'_{M',g}(v,\gamma) \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),S\cup\{g\}} \nu^{\mathscr{F}}_{\Sigma,S,g}(\downarrow s\,.\,\varrho)
\end{array}$$

Case $(@^F s)\varrho$:

$$\begin{array}{ll}
& \nu^{\text{Mod}}_{\Sigma}(M'), v, \gamma \models^{\text{EDHML}}_{\Sigma,S} (@^F s)\varrho \\
\Leftrightarrow & \{\text{ def. } \models^{\text{EDHML}} \} \\
& \nu^{\text{Mod}}_{\Sigma}(M'), v, \gamma' \models^{\text{EDHML}}_{\Sigma,S} \varrho \\
& \qquad \text{for all } \gamma' \in \Gamma^F(\nu^{\text{Mod}}_{\Sigma}(M')) \text{ with } c(\nu^{\text{Mod}}_{\Sigma}(M'))(\gamma') = v(s) \\
\Leftrightarrow & \{\text{ I.H. } \} \\
& M', \beta'_{M',g'}(v,\gamma') \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),S\cup\{g'\}} \nu^{\mathscr{F}}_{\Sigma,S,g'}(\varrho) \\
& \qquad \text{for all } \gamma' \in \Gamma^F(\nu^{\text{Mod}}_{\Sigma}(M')) \text{ with } c(\nu^{\text{Mod}}_{\Sigma}(M'))(\gamma') = v(s) \\
\Leftrightarrow & \{\text{ def. } \models^{\text{CASL}} \} \\
& M', \beta'_{M',g}(v,\gamma) \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),S\cup\{g\}} \\
& \qquad \forall g' : \text{Conf}\,.\,(\text{c}(g') = s \wedge \text{reachable}(F,g')) \Rightarrow \nu^{\mathscr{F}}_{\Sigma,S,g'}(\varrho) \\
\Leftrightarrow & \{\text{ def. } \nu^{\mathscr{F}} \} \\
& M', \beta'_{M',g}(v,\gamma) \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),S\cup\{g\}} \nu^{\mathscr{F}}_{\Sigma,S,g}((@^F s)\varrho)
\end{array}$$

Case $\Box^F\varrho$:

$$\begin{array}{ll}
& \nu^{\text{Mod}}_{\Sigma}(M'), v, \gamma \models^{\text{EDHML}}_{\Sigma,S} \Box^F\varrho \\
\Leftrightarrow & \{\text{ def. } \models^{\text{EDHML}} \} \\
& \nu^{\text{Mod}}_{\Sigma}(M'), v, \gamma' \models^{\text{EDHML}}_{\Sigma,S} \varrho \quad \text{for all } \gamma' \in \Gamma^F(\nu^{\text{Mod}}_{\Sigma}(M'), \gamma) \\
\Leftrightarrow & \{\text{ I.H. } \} \\
& M', \beta'_{M',g'}(v,\gamma') \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),S\cup\{g'\}} \nu^{\mathscr{F}}_{\Sigma,S,g'}(\varrho) \quad \text{for all } \gamma' \in \Gamma^F(\nu^{\text{Mod}}_{\Sigma}(M'), \gamma) \\
\Leftrightarrow & \{\text{ def. } \models^{\text{CASL}} \} \\
& M', \beta'_{M',g}(v,\gamma) \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),S\cup\{g\}} \forall g' : \text{Conf}\,.\,\text{reachable}(F,g,g') \Rightarrow \nu^{\mathscr{F}}_{\Sigma,S,g'}(\varrho)
\end{array}$$

$\Leftrightarrow \quad \{$ def. $\nu^{\mathscr{F}} \}$
$M', \beta'_{M',g}(v,\gamma) \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),S\cup\{g\}} \nu^{\mathscr{F}}_{\Sigma,S,g}(\Box^F \varrho)$

Case $\langle e(X) /\!\!/ \psi \rangle \varrho$:

$\nu^{\text{Mod}}_\Sigma(M'), v, \gamma \models^{\text{EDHML}}_{\Sigma,S} \langle e(X) /\!\!/ \psi \rangle \varrho$
$\Leftrightarrow \quad \{$ def. $\models^{\text{EDHML}} \}$
$\nu^{\text{Mod}}_\Sigma(M'), v, \gamma \models^{\text{EDHML}}_{\Sigma,S} \varrho$
for some $\beta : X \to \mathcal{D}$, $\gamma' \in \Gamma(\nu^{\text{Mod}}_\Sigma(M'))$ with
$(\gamma,\gamma') \in R(\nu^{\text{Mod}}_\Sigma(M'))_{e(\beta)}$ and
$(\omega(\nu^{\text{Mod}}_\Sigma(M'))(\gamma), \omega(\nu^{\text{Mod}}_\Sigma(M'))(\gamma')), \beta \models^{2\mathcal{D}}_{A(\Sigma),X} \psi$
$\Leftrightarrow \quad \{$ I. H., def. $\models^{2\mathcal{D}}$, def. $\models^{\text{CASL}} \}$
$M', \beta'_{M',g}(v,\gamma) \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),S\cup\{g\}} \nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho)$
for some $\beta : X \to \mathcal{D}$, $\gamma' \in \Gamma(\nu^{\text{Mod}}_\Sigma(M'))$ with
$(\gamma,\gamma') \in R(\nu^{\text{Mod}}_\Sigma(M'))_{e(\beta)}$ and
$M', \iota^{-1}_{M',dt} \circ ((\omega(\nu^{\text{Mod}}_\Sigma(M'))(\gamma) + \omega(\nu^{\text{Mod}}_\Sigma(M'))(\gamma')) \cup \beta) \cup$
$\{g \mapsto \iota_{M',\text{Conf}}(\gamma), g' \mapsto \iota_{M',\text{Conf}}(\gamma')\} \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),X\cup\{g,g'\}}$
$\mathscr{F}^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),A(\Sigma)(g)\cup A(\Sigma)(g')\cup 1_X}(\psi)$
$\Leftrightarrow \quad \{$ def. $\models^{\text{CASL}} \}$
$M', \beta'_{M',g}(v,\gamma) \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),S\cup\{g\}}$
$\exists X : dt \,.\, \exists g' : \text{Conf} \,.\, \text{trans}(g, e(X), g') \wedge$
$\mathscr{F}^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),A(\Sigma)(g)\cup A(\Sigma)(g')\cup 1_X}(\psi) \wedge \nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho)$
$\Leftrightarrow \quad \{$ def. $\nu^{\mathscr{F}} \}$
$M', \beta'_{M',g}(v,\gamma) \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),S\cup\{g\}} \nu^{\mathscr{F}}_{\Sigma,S,g}(\langle e(X) /\!\!/ \psi \rangle \varrho)$

Case $\langle\!| e(X) : \phi /\!\!/ \psi |\!\rangle \varrho$:

$\nu^{\text{Mod}}_\Sigma(M'), v, \gamma \models^{\text{EDHML}}_{\Sigma,S} \langle\!| e(X) : \phi /\!\!/ \psi |\!\rangle \varrho$
$\Leftrightarrow \quad \{$ def. $\models^{\text{EDHML}} \}$
$\nu^{\text{Mod}}_\Sigma(M'), v, \gamma \models^{\text{EDHML}}_{\Sigma,S} \varrho$
for all $\beta : X \to \mathcal{D}$ such that $\omega(\nu^{\text{Mod}}_\Sigma(M'))(\gamma), \beta \models^{\mathcal{D}}_{A(\Sigma),X} \phi$
and some $\gamma' \in \Gamma(\nu^{\text{Mod}}_\Sigma(M'))$ with $(\gamma,\gamma') \in R(\nu^{\text{Mod}}_\Sigma(M'))_{e(\beta)}$
and $(\omega(\nu^{\text{Mod}}_\Sigma(M'))(\gamma), \omega(\nu^{\text{Mod}}_\Sigma(M'))(\gamma')), \beta \models^{2\mathcal{D}}_{A(\Sigma),X} \psi$
$\Leftrightarrow \quad \{$ I. H., def. $\models^{\mathcal{D}}$, def. $\models^{2\mathcal{D}}$, def. $\models^{\text{CASL}} \}$
$M', \beta'_{M',g}(v,\gamma) \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),S\cup\{g\}} \nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho)$

$$
\begin{array}{l}
\quad \text{for all } \beta : X \to \mathcal{D} \text{ such that} \\
\quad\quad M', \iota^{-1}_{M',dt} \circ (\omega(\nu^{\mathrm{Mod}}_{\Sigma}(M'))(\gamma) \cup \beta) \cup \\
\quad\quad\quad \{g \mapsto \iota_{M',\mathrm{Conf}}(\gamma)\} \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma), X \cup \{g\}} \mathscr{F}^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma), A(\Sigma)(g) \cup 1_X}(\phi) \\
\quad \text{and some } \gamma' \in \Gamma(\nu^{\mathrm{Mod}}_{\Sigma}(M')) \text{ with } (\gamma, \gamma') \in R(\nu^{\mathrm{Mod}}_{\Sigma}(M'))_{e(\beta)} \\
\quad \text{and } M', \iota^{-1}_{M',dt} \circ ((\omega(\nu^{\mathrm{Mod}}_{\Sigma}(M'))(\gamma) + \omega(\nu^{\mathrm{Mod}}_{\Sigma}(M'))(\gamma')) \cup \beta) \cup \\
\quad\quad\quad \{g \mapsto \iota_{M',\mathrm{Conf}}(\gamma), g' \mapsto \iota_{M',\mathrm{Conf}}(\gamma')\} \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma), X \cup \{g,g'\}} \\
\quad\quad\quad\quad \mathscr{F}^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma), A(\Sigma)(g) \cup A(\Sigma)(g') \cup 1_X}(\psi) \\
\Leftrightarrow \quad \{ \text{ def. } \models^{\mathrm{CASL}} \} \\
\quad M', \beta'_{M',g}(v, \gamma) \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma), S \cup \{g\}} \\
\quad\quad \forall X : dt . \mathscr{F}^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma), A(\Sigma)(g) \cup 1_X}(\phi) \Rightarrow \\
\quad\quad\quad \exists g' : \mathrm{Conf} . \mathrm{trans}(g, e(X), g') \wedge \\
\quad\quad\quad\quad \mathscr{F}^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma), A(\Sigma)(g) \cup A(\Sigma)(g') \cup 1_X}(\psi) \wedge \nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho) \\
\Leftrightarrow \quad \{ \text{ def. } \nu^{\mathscr{F}} \} \\
\quad M', \beta'_{M',g}(v, \gamma) \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma), S \cup \{g\}} \nu^{\mathscr{F}}_{\Sigma,S,g}(\langle\!| e(X) : \phi /\!/ \psi |\!\rangle \varrho)
\end{array}
$$

Case $\neg\varrho$:

$$
\begin{array}{l}
\quad \nu^{\mathrm{Mod}}_{\Sigma}(M'), v, \gamma \models^{\mathrm{EDHML}}_{\Sigma,S} \neg\varrho \\
\Leftrightarrow \quad \{ \text{ def. } \models^{\mathrm{EDHML}} \} \\
\quad \nu^{\mathrm{Mod}}_{\Sigma}(M'), v, \gamma \not\models^{\mathrm{EDHML}}_{\Sigma,S} \varrho \\
\Leftrightarrow \quad \{ \text{ I.H. } \} \\
\quad M', \beta'_{M',g}(v, \gamma) \not\models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma), S \cup \{g\}} \nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho) \\
\Leftrightarrow \quad \{ \text{ def. } \models^{\mathrm{CASL}} \} \\
\quad M', \beta'_{M',g}(v, \gamma) \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma), S \cup \{g\}} \neg(\nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho)) \\
\Leftrightarrow \quad \{ \text{ def. } \nu^{\mathscr{F}} \} \\
\quad M', \beta'_{M',g}(v, \gamma) \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma), S \cup \{g\}} \nu^{\mathscr{F}}_{\Sigma,S,g}(\neg\varrho)
\end{array}
$$

Case $\varrho \vee \varsigma$:

$$
\begin{array}{l}
\quad \nu^{\mathrm{Mod}}_{\Sigma}(M'), v, \gamma \models^{\mathrm{EDHML}}_{\Sigma,S} \varrho \vee \varsigma \\
\Leftrightarrow \quad \{ \text{ def. } \models^{\mathrm{EDHML}} \} \\
\quad \nu^{\mathrm{Mod}}_{\Sigma}(M'), v, \gamma \models^{\mathrm{EDHML}}_{\Sigma,S} \varrho \text{ or } \nu^{\mathrm{Mod}}_{\Sigma}(M'), v, \gamma \models^{\mathrm{EDHML}}_{\Sigma,S} \varsigma \\
\Leftrightarrow \quad \{ \text{ I.H. } \} \\
\quad M', \beta'_{M',g}(v, \gamma) \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma), S \cup \{g\}} \nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho) \text{ or } M', \beta'_{M',g}(v, \gamma) \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma), S \cup \{g\}} \nu^{\mathscr{F}}_{\Sigma,S,g}(\varsigma) \\
\Leftrightarrow \quad \{ \text{ def. } \models^{\mathrm{CASL}} \} \\
\quad M', \beta'_{M',g}(v, \gamma) \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma), S \cup \{g\}} (\nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho)) \vee (\nu^{\mathscr{F}}_{\Sigma,S,g}(\varsigma)) \\
\Leftrightarrow \quad \{ \text{ def. } \nu^{\mathscr{F}} \}
\end{array}
$$

$$M', \beta'_{M',g}(v,\gamma) \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),S\cup\{g\}} \nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho \vee \varsigma)$$

□

Theorem 8.3 $(\nu^{\text{Sig}}, \nu^{\text{Mod}}, \nu^{\text{Sen}})$ is a theoroidal comorphism from EDHML to CASL.

Proof. Let $\Sigma \in \text{Sig}^{\text{EDHML}}$, $M' \in |\text{Mod}^{\text{CASL}}(\nu^{\text{Sig}}(\Sigma))|$, and $\rho \in \text{Sen}^{\text{EDHML}}(\Sigma)$. The satisfaction condition follows from

$$\begin{aligned}
& \nu^{\text{Mod}}_{\Sigma}(M') \models^{\text{EDHML}}_{\Sigma} \rho \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\text{EDHML}} \} \\
& \nu^{\text{Mod}}_{\Sigma}(M'), \emptyset, \gamma_0 \models^{\text{EDHML}}_{\Sigma,\emptyset} \rho \quad \text{for all } \gamma_0 \in \Gamma_0(\nu^{\text{Mod}}_{\Sigma}(M')) \\
\Leftrightarrow \quad & \{ \text{ Lem. 8.2 } \} \\
& M', \beta'_{M',g}(\emptyset, \gamma_0) \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),\{g\}} \nu^{\mathscr{F}}_{\Sigma,\emptyset,g}(\rho) \quad \text{for all } \gamma_0 \in \Gamma_0(\nu^{\text{Mod}}_{\Sigma}(M')) \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\text{CASL}} \} \\
& M' \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma)} \forall g : \text{Conf}.\text{init}(g) \Rightarrow \nu^{\mathscr{F}}_{\Sigma,\emptyset,g}(\rho) \\
\Leftrightarrow \quad & \{ \text{ def. } \nu^{\text{Sen}} \} \\
& M' \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma)} \nu^{\text{Sen}}_{\Sigma}(\rho)
\end{aligned}$$

□

Chapter 9

EDHMLO: Extending EDHML with Outputs

Contents

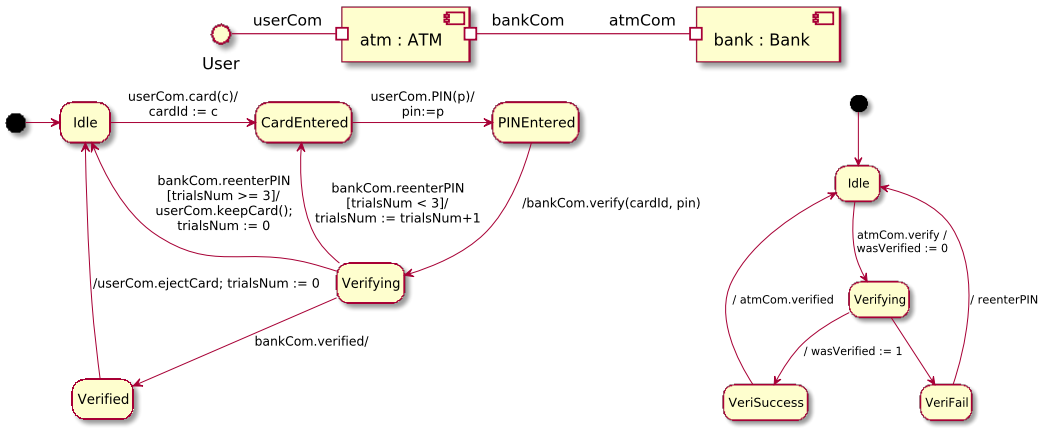


Figure 9.1: UML diagrams for the `ATM` example verified in Sect. 13.2 (implicit completion events omitted): Composite Structure diagram: top; State Machines: left `ATM`, right `Bank`.

In this chapter, we extend the institution and State Machine language from Chapter 7 with output events, thereby preparing the State Machines for Communication in the context of a Composite Structure. As a motivating example, Fig. 9.1 repeats the ATM system from the introduction, for we include a full specification parsable by our implementation in App. C and prove a safety property in Chapter 13.

In our architecture of communication, the queues are removed from machine configurations and treated instead in the Composite Structures. In the State Machines themselves, communication with the outside needs no handling beyond mentioning the messages in transitions; the existence of ports can largely be ignored by machines. This architecture means that, to handle communication on the State Machine side, we need only add output events. However, doing this in a way that respects the satisfaction condition is again somewhat involved, in particular, expressing the sets of input and output events in a context without full higher-order logic.

The contents of this chapter have been published in similar form as part of [RKR22].

The construction of EDHMLO was a common effort between Alexander Knapp, Markus Roggenbach and myself. In our virtual meetings, we jointly sketched the constructions, proofs and examples, which I then worked out in detail and wrote down in their final form.

9.1 The Underlying Data Universe

We assume a consistent, monomorphic CASL specification Dt of *data*. The interpretation of the sorts $S(Dt)$ of Dt represents the different kinds of data. In EDHML we had no need for such a type system. We do need it now because later, when we flatten Composite Structures, it will have to be possible for an attribute to contain a message queue or an entire submachine configuration. Requiring Dt to be monomorphic fixes the carrier sets as there is, up to isomorphism, a single model $\mathcal{D}$ of Dt. We also use open formulæ $\mathscr{F}^{\text{CASL}}_{\text{Sig}(Dt),X}$ over sorted variables $X = (X_s)_{s\in S(Dt)}$ and their satisfaction relation $\mathcal{D},\beta \models^{\text{CASL}}_{\text{Sig}(Dt),X} \varphi$ for a variable valuation $\beta : X \to \mathcal{D}$, i.e., $\beta = (\beta_s : X_s \to s^{\mathcal{D}})_{s\in S(Dt)}$.

Example 9.1 In our implementation, the State Machines themselves still use data signatures containing the natural numbers as the sole and implicit sort (richer data signatures will be needed to encode Composite Structures as Event/Data systems). A suitable specification for natural numbers would be the one from Fig. 4.1. The guards from Fig. 9.1 are examples of the open formulæ. The formulæ are open because they can contain both attributes and variables bound by the trigger specification. An example valuation

$$\{\text{trialsNum} \mapsto 2; \text{cardId} \mapsto 0; \text{pin} \mapsto 42\}$$

would not satisfy the formula $\text{trialsNum} \geq 3$, whereas the valuation

$$\{\text{trialsNum} \mapsto 6; \text{cardId} \mapsto 0; \text{pin} \mapsto 2\}$$

would.

9.2 Data States and Transitions

A *data signature* A consists of a finite set of *attributes* $|A|$ and a sorting $s(A) : |A| \to S(Dt)$. A *data signature morphism* from a data signature A to a data signature A' is a function $\alpha : |A| \to |A'|$ such that $s(A)(a) = s(A')(\alpha(a))$ for all $a \in |A|$. We sometimes identify A with the $S(Dt)$-sorted family $(s(A)^{-1}(s))_{s \in S(Dt)}$.

Example 9.2 A data signature for the ATM machine in Fig. 9.1 could be given by

$$\begin{aligned} |A| &= \{\mathrm{trialsNum}, \mathrm{cardId}, \mathrm{pin}\} \\ s(A)(x) &= \mathrm{Nat} \quad \text{for all } x \end{aligned}$$

A *data state* ω for a data signature A is given by an attribute valuation $\omega : A \to \mathcal{D}$, i.e., $\omega(a) \in s(A)(a)^{\mathcal{D}}$ for $a \in |A|$; in particular, $\Omega(A) = \mathcal{D}^A$ is the set of A-data states. The *state predicates* $\mathscr{F}^{\mathcal{D}}_{A,X}$ are the formulæ in $\mathscr{F}^{\textsc{Casl}}_{\mathrm{Sig}(Dt),A \cup X}$, taking A as well as an additional $S(Dt)$-indexed family X as variables. A state predicate $\phi \in \mathscr{F}^{\mathcal{D}}_{A,X}$ is to be interpreted over an A-data state ω and variable valuation $\beta : X \to \mathcal{D}$ and we define the *satisfaction relation* $\models^{\mathcal{D}}$ by

$$\omega, \beta \models^{\mathcal{D}}_{A,X} \phi \iff \mathcal{D}, \omega \cup \beta \models^{\textsc{Casl}}_{\mathrm{Sig}(Dt),A \cup X} \phi \,.$$

The *α-reduct* of an A'-data state $\omega' : A' \to \mathcal{D}$ along a data signature morphism $\alpha : A \to A'$ is given by the A-data state $\omega'|\alpha : A \to \mathcal{D}$ with $(\omega'|\alpha)(a) = \omega'(\alpha(a))$ for every $a \in |A|$. The *state predicate translation* $\mathscr{F}^{\mathcal{D}}_{\alpha,X} : \mathscr{F}^{\mathcal{D}}_{A,X} \to \mathscr{F}^{\mathcal{D}}_{A',X}$ along $\alpha : A \to A'$ is given by the CASL-formula translation $\mathscr{F}^{\textsc{Casl}}_{\mathrm{Sig}(Dt),\alpha \cup 1_X}$ along the substitution $\alpha \cup 1_X$. Reduct and translation fulfil the following *satisfaction condition* due to the general substitution lemma for CASL:

$$\omega'|\alpha, \beta \models^{\mathcal{D}}_{A,X} \phi \iff \omega', \beta \models^{\mathcal{D}}_{A',X} \mathscr{F}^{\mathcal{D}}_{\alpha,X}(\phi) \,.$$

A *data transition* (ω, ω') for a data signature A is a pair of A-data states; in particular, $\Omega^2(A) = (\mathcal{D}^A)^2$ is the set of A-data transitions. It holds that $(\mathcal{D}^A)^2 \cong \mathcal{D}^{2A}$, where $2A = A \uplus A$ and we assume that no attribute in A ends in a prime $'$ and all attributes in the second summand are adorned with an additional prime. The *transition predicates* $\mathscr{F}^{2\mathcal{D}}_{A,X}$ are the formulæ $\mathscr{F}^{\mathcal{D}}_{2A,X}$. We will write $\mathrm{cl}_A(\psi)$ for the transition predicate asserting that ψ holds and that all attributes $a \in A$ not mentioned by ψ remain unchanged.

The satisfaction relation $\models^{2\mathcal{D}}$ for a transition predicate $\psi \in \mathscr{F}^{2\mathcal{D}}_{A,X}$, data transition $(\omega, \omega') \in \Omega^2(A)$, and valuation $\beta : X \to \mathcal{D}$ is defined as

$$(\omega, \omega'), \beta \models^{2\mathcal{D}}_{A,X} \psi \iff \omega + \omega', \beta \models^{\mathcal{D}}_{2A,X} \psi$$

where $\omega + \omega' \in \Omega(2A)$ with $(\omega + \omega')(a) = \omega(a)$ and $(\omega + \omega')(a') = \omega'(a)$.

The *α-reduct* of an A'-data transition (ω', ω'') along a data signature morphism $\alpha : A \to A'$ is given by the A-data transition $(\omega', \omega'')|\alpha = (\omega'|\alpha, \omega''|\alpha)$. The *transition predicate translation* $\mathscr{F}^{2\mathcal{D}}_{\alpha,X}$ along α by $\mathscr{F}^{\mathcal{D}}_{2\alpha,X}$ with $2\alpha : 2A \to 2A'$ defined by $2\alpha(a) = \alpha(a)$ and $2\alpha(a') = \alpha(a)'$. Like for data states, reduct and translation fulfil the following *satisfaction condition*:

$$(\omega', \omega'')|\alpha, \beta \models^{2\mathcal{D}}_{A,X} \psi \iff (\omega', \omega''), \beta \models^{2\mathcal{D}}_{A',X} \mathscr{F}^{2\mathcal{D}}_{\alpha,X}(\psi) \,.$$

9.3 Events and Messages

An *event signature* E consists of a finite set of events $|E|$ and a map $\overline{s}(E) : |E| \to S(Dt)^*$ assigning to each $e \in |E|$ its list of parameter sorts. An *event signature morphism* $\eta : E \to E'$ is a function $\eta : |E| \to |E'|$ such that $\overline{s}(E)(e) = \overline{s}(E')(\eta(e))$ for all $e \in |E|$. We write $e(X)$ for $e \in |E|$ and $\overline{s}(E)(e) = s_1, \ldots, s_n$ when choosing n different *parameters* $X = x_1, \ldots, x_n$, and also $e(X) \in E$ in this case; when $f = e(X)$, we write $X(f)$ for X and we furthermore lift this notation to sets and lists of events. We sometimes identify the parameter list X with the $S(Dt)$-sorted family $(\{x_i \mid s_i = s\})_{s \in S(Dt)}$ and write $\overline{s}(E)(e)(x_i)$ for s_i.

A *message* $e(\beta)$ over an event signature E is given by an event $e(X) \in E$ with its parameters X instantiated by a parameter valuation $\beta : X \to \mathcal{D}$ such that $\beta(x) \in s^{\mathcal{D}}$ for $\overline{s}(E)(e)(x) = s$; the set of all messages over an event signature E is denoted by $\hat{E}(E)$. When $\hat{e} = e(\beta) \in \hat{E}(E)$, we write $\beta(\hat{e})$ for β, and when $e(X) \in E$ and $\beta : Y \to \mathcal{D}$ for $X \subseteq Y$, we write $e(\beta)$ for $e(\beta{\restriction}X)$; both notations are furthermore lifted to sets and lists of messages.

Example 9.3 An event signature for the output events of the ATM machine in Fig. 9.1 could be given by:

$$
\begin{aligned}
|E| &= \{\mathsf{bankCom.verify}, \mathsf{userCom.ejectCard}, \mathsf{userCom.keepCard}\} \\
\overline{s}(E)(\mathsf{bankCom.verify}) &= (\mathsf{Nat}, \mathsf{Nat}) \\
\overline{s}(E)(\mathsf{userCom.ejectCard}) &= () \\
\overline{s}(E)(\mathsf{userCom.keepCard}) &= ()
\end{aligned}
$$

Note that these events contain port names (before the dot) in a way that is transparent to the formalism. This will be become important when we discuss Composite Structures. In our example signature, we have one event which takes two parameters of sort Nat and two events which take none. A message could then be given by $(\mathsf{bankCom.verify}, (7,1))$, which we write more conveniently as $\mathsf{bankCom.verify}(7,1)$.

In order to mimic the UML 'ignore operator', we define the set of *shuffling* $\hat{F}_1 \parallel \hat{F}_2$ of two message lists $\hat{F}_1$ and $\hat{F}_2$ inductively by

$$
\begin{aligned}
&\hat{F} \parallel \varepsilon = \{\hat{F}\} \\
=&\varepsilon \parallel \hat{F}\,, \quad (\hat{f} :: \hat{F}_1) \parallel \hat{F}_2 \\
=&\{\hat{f} :: \hat{F} \mid \hat{F} \in \hat{F}_1 \parallel \hat{F}_2\} \\
=&\hat{F}_1 \parallel (\hat{f} :: \hat{F}_2)\,.
\end{aligned}
$$

An event signature morphism $\eta : E \to E'$ is lifted to a message $e(\beta) \in \hat{E}(E)$ by setting $\hat{E}(\eta)(e(\beta)) = \eta(e)(\beta) \in \hat{E}(E')$ and also to sets and lists of messages.

9.4 Event/Data Signatures

An *event/data signature* Σ consists of *input* and *output* event signatures $I(\Sigma)$ and $O(\Sigma)$, and a data signature $A(\Sigma)$. An *event/data signature morphism* $\sigma : \Sigma \to \Sigma'$ consists of an

input event signature morphism $I(\sigma) : I(\Sigma) \to I(\Sigma')$, an output event signature morphism $O(\sigma) : O(\Sigma) \to O(\Sigma')$, and a data signature morphism $A(\sigma) : A(\Sigma) \to A(\Sigma')$. We lift the event signatures and signature morphisms to messages by writing $\hat{I}(\Sigma)$ for $\hat{E}(I(\Sigma))$, $\hat{O}(\Sigma)$ for $\hat{E}(O(\Sigma))$, $\hat{I}(\sigma)$ for $\hat{E}(I(\sigma))$, and $\hat{O}(\sigma)$ for $\hat{E}(O(\sigma))$.

The category of EDHMLO-*signatures* $\mathrm{Sig}^{\mathrm{EDHMLO}}$ consists of the event/data signatures and signature morphisms.

9.5 Event/Data Structures

A *configuration* $\gamma = (c, d)$ consists of

– a *control state* c from some set of control states C and

– a *data state* d from some set of data states D.

Given a data signature A the data state of γ may be *labelled* by a map ω such that $\omega(d) \in \Omega(A)$. For a set of configurations Γ we write $C(\Gamma)$ for its set of control states.

A Σ-*event/data structure* $M = (\Gamma, R, \Gamma_0, \omega)$ over an event/data signature Σ consists of

– a set of *configurations* $\Gamma \subseteq C \times D$,

– a family of *transition relations* $R = (R_{\hat{\imath},\hat{O}} \subseteq \Gamma \times \Gamma)_{\hat{\imath} \in \hat{I}(\Sigma), \hat{O} \in \hat{O}(\Sigma)^*}$, and

– a non-empty set of *initial configurations* $\Gamma_0 \subseteq \Gamma$

such that Γ is *reachable* from Γ_0 via R, i.e., for all $\gamma \in \Gamma$ there are $\gamma_0 \in \Gamma_0$, $n \geq 0$, $\hat{\imath}_1, \ldots, \hat{\imath}_n \in \hat{I}(\Sigma)$, $\hat{O}_1, \ldots, \hat{O}_n \in \hat{O}(\Sigma)^*$, and $(\gamma_k, \gamma_{k+1}) \in R_{\hat{\imath}_{k+1},\hat{O}_{k+1}}$ for all $0 \leq k < n$ with $\gamma_n = \gamma$; and a *data state labelling* $\omega : D \to \Omega(A(\Sigma))$. We write $c(M)(\gamma) = c$ and $\omega(M)(\gamma) = \omega(d)$ for $\gamma = (c, d) \in \Gamma$, $\Gamma(M)$ for Γ, $C(M)$ for $\{c(M)(\gamma) \mid \gamma \in \Gamma(M)\}$, $R(M)$ for R, $\Gamma_0(M)$ for Γ_0, $C_0(M)$ for $C(\Gamma_0)$, and $\Omega_0(M)$ for $\{\omega(M)(\gamma_0) \mid \gamma_0 \in \Gamma_0\}$.

The above definition restricts structures to reachable ones only. This corresponds to our upcoming definition of the EDHMLO logic. In this logic, sentences will hold in a structure if they are satisfied in its initial states. However, using modalities, we still want to be able to express that a certain property holds for all states.

The σ-*reduct* of a Σ'-event/data structure M' along the event/data signature morphism $\sigma : \Sigma \to \Sigma'$ is the Σ-event/data structure $M'|\sigma$ such that

– $\Gamma(M'|\sigma) \subseteq \Gamma(M')$ as well as $R(M'|\sigma) = (R(M'|\sigma)_{\hat{\imath},\hat{O}})_{\hat{\imath} \in \hat{I}(\Sigma), \hat{O} \in \hat{O}(\Sigma)^*}$ are inductively defined by $\Gamma_0(M') \subseteq \Gamma(M'|\sigma)$ and, for all $\gamma', \gamma'' \in \Gamma(M')$, $\hat{\imath} \in \hat{I}(\Sigma)$, and $\hat{O} \in \hat{O}(\Sigma)^*$, if $\gamma' \in \Gamma(M'|\sigma)$ and $(\gamma', \gamma'') \in R(M')_{\hat{I}(\sigma)(\hat{\imath}),\hat{O}(\sigma)(\hat{O})}$, then $\gamma'' \in \Gamma(M'|\sigma)$ and $(\gamma', \gamma'') \in R(M'|\sigma)_{\hat{\imath},\hat{O}}$;

– $\Gamma_0(M'|\sigma) = \Gamma_0(M')$; and

– $\omega(M'|\sigma)(\gamma') = (\omega(M')(\gamma'))|A(\sigma)$ for all $\gamma' \in \Gamma(M'|\sigma)$.

This σ-reduct keeps exactly those transitions that are a direct image along σ. It would also be possible to additionally keep transitions that show a super-list of the outputs that can be reached by σ. When moving to EDHMLO sentences, however, it turns out to be impossible to fix a particular list of outputs.

Given sets of input events $J \subseteq I(\Sigma)$ and output events $N \subseteq O(\Sigma)$, we denote by $\Gamma^{J,N}(M,\gamma)$ and $\Gamma^{J,N}(M)$, respectively, the set of configurations of a Σ-event/data structure M that are J,N-reachable from a configuration $\gamma \in \Gamma(M)$ and from an initial configuration $\gamma_0 \in \Gamma_0(M)$, respectively. Here a $\gamma_n \in \Gamma(M)$ is *J,N-reachable in M from* a $\gamma_1 \in \Gamma(M)$ if there are $n \geq 1$, $\hat{\imath}_2,\ldots,\hat{\imath}_n \in \hat{I}(J)$, $\hat{O}_2,\ldots,\hat{O}_n \in \hat{O}(N)^*$, and $(\gamma_i,\gamma_{i+1}) \in R(M)_{\hat{\imath}_{k+1},\hat{O}_{k+1}}$ for all $1 \leq k < n$.

The Σ-event/data structures form the discrete category $\mathrm{Mod}^{\mathrm{EDHMLO}}(\Sigma)$ of EDHMLO *structures* over Σ. For each $\sigma : \Sigma \to \Sigma'$ in $\mathrm{Sig}^{\mathrm{EDHMLO}}$ the *σ-reduct functor* $\mathrm{Mod}^{\mathrm{EDHMLO}}(\sigma) : \mathrm{Mod}^{\mathrm{EDHMLO}}(\Sigma') \to \mathrm{Mod}^{\mathrm{EDHMLO}}(\Sigma)$ is given by $\mathrm{Mod}^{\mathrm{EDHMLO}}(\sigma)(M') = M'|\sigma$.

9.6 Event/Data Formulæ and Sentences

The Σ-*event/data formulæ* $\mathscr{F}^{\mathrm{EDHMLO}}_{\Sigma,S}$ over an event/data signature Σ and a set of *state variables* S are inductively defined by

- φ — data state sentence $\varphi \in \mathscr{F}^{\mathcal{D}}_{A(\Sigma),\emptyset}$ holds in the current configuration;
- s — the control state of the current configuration is $s \in S$ (unchanged compared to EDHML);
- $\downarrow s\,.\,\varrho$ — calling the current control state s, formula $\varrho \in \mathscr{F}^{\mathrm{EDHMLO}}_{\Sigma,S\uplus\{s\}}$ holds; note that s is turned into a fresh variable by adding it with the disjoint sum operator to the set of state variables (unchanged compared to EDHML);
- $(@^{J,N}s)\varrho$ — in all configurations with control state $s \in S$ that are J,N-reachable, formula $\varrho \in \mathscr{F}^{\mathrm{EDHMLO}}_{\Sigma,S}$ holds (compared to EDHML, the output events N have been added here);
- $\Box^{J,N}\varrho$ — in all configurations that are J,N-reachable from the current configuration formula $\varrho \in \mathscr{F}^{\mathrm{EDHMLO}}_{\Sigma,S}$ holds ('jump' operator; compared to EDHML, the output events N have been added here);
- $\Diamond i[O]_N\psi\varrho$ — in the current configuration there are valuations $\beta : X(i) \to \mathcal{D}$, $\beta' : X(O) \to \mathcal{D}$, and a transition for the incoming message $i(\beta) \in \hat{I}(\Sigma)$ and the outgoing messages $\hat{O}' \in O(\beta') \parallel \hat{N}$ for $O(\beta') \in \hat{O}(\Sigma)^*$, $\hat{N} \in \hat{O}(N)^*$ such that $\beta \cup \beta'$ satisfies transition formula $\psi \in \mathscr{F}^{2\mathcal{D}}_{A(\Sigma),X(i)\cup X(O)}$ and $\varrho \in \mathscr{F}^{\mathrm{EDHMLO}}_{\Sigma,S}$ holds afterwards (compared to EDHML, the stated (O) and ignored (N) output messages have been added);
- $\langle\!|i : \phi /\!/ [O]_N : \psi|\!\rangle\varrho$ — in the current configuration for all valuations $\beta : X(i) \to \mathcal{D}$ satisfying state formula $\phi \in \mathscr{F}^{\mathcal{D}}_{A(\Sigma),X(i)}$ there are a valuation $\beta' : X(O) \to \mathcal{D}$ and a transition for the incoming message $i(\beta) \in \hat{I}(\Sigma)$ and the outgoing messages $\hat{O}' \in O(\beta') \parallel \hat{N}$ for $O(\beta') \in \hat{O}(\Sigma)^*$, $\hat{N} \in \hat{O}(N)^*$ such that $\beta \cup \beta'$ satisfies transition formula $\psi \in \mathscr{F}^{2\mathcal{D}}_{A(\Sigma),X(i)\cup X(O)}$ and $\varrho \in \mathscr{F}^{\mathrm{EDHMLO}}_{\Sigma,S}$ holds afterwards (compared to EDHML, the stated (O) and ignored (N) output messages have been added);

– $\neg\varrho$ — in the current configuration $\varrho \in \mathscr{F}^{\mathrm{EDHMLO}}_{\Sigma,S}$ does not hold (unchanged compared to EDHML);

– $\varrho_1 \vee \varrho_2$ — in the current configuration $\varrho_1 \in \mathscr{F}^{\mathrm{EDHMLO}}_{\Sigma,S}$ or $\varrho_2 \in \mathscr{F}^{\mathrm{EDHMLO}}_{\Sigma,S}$ holds (unchanged compared to EDHML).

We write $(@s)\varrho$ for $(@^{I(\Sigma),O(\Sigma)}s)\varrho$, $\Box\varrho$ for $\Box^{I(\Sigma),O(\Sigma)}\varrho$, $\Box i[O]_N\psi\varrho$ for $\neg\Diamond i[O]_N\psi\neg\varrho$, and true for $\downarrow s\,.\,s$; we write O for $[O]_\emptyset$.

Two different kinds of relativisations are used in EDHMLO-formulæ: For the jump operator $(@^{J,N}s)\varrho$ and the always operator $\Box^{J,N}\varrho$ the subsets of input events $J \subseteq I(\Sigma)$ and output events $N \subseteq O(\Sigma)$ restrict the referrable states in an EDHMLO-structure to those that are J,N-reachable. Note that this information can be kept through translations to another EDHMLO-signature. On the other hand, $[O]_N$ specifies that besides messages from O additional messages for events in $N \subseteq O(\Sigma)$ can be mixed into the output. However, in particular, $[O]_\emptyset$ requires exactly O. The purpose of the relativisation $[O]_N$ is to specify with finitely many formulae (the set of output events is finite) to ignore message lists of arbitrary length.

Let $\sigma : \Sigma \to \Sigma'$ be an event/data signature morphism. The *event/data formulæ translation* $\mathscr{F}^{\mathrm{EDHMLO}}_{\sigma,S} : \mathscr{F}^{\mathrm{EDHMLO}}_{\Sigma,S} \to \mathscr{F}^{\mathrm{EDHMLO}}_{\Sigma',S}$ along σ is recursively given by

– $\mathscr{F}^{\mathrm{EDHMLO}}_{\sigma,S}(\varphi) = \mathscr{F}^{\mathcal{D}}_{A(\sigma),\emptyset}(\varphi)$;

– $\mathscr{F}^{\mathrm{EDHMLO}}_{\sigma,S}(s) = s$;

– $\mathscr{F}^{\mathrm{EDHMLO}}_{\sigma,S}(\downarrow s\,.\,\varrho) = \downarrow s\,.\,\mathscr{F}^{\mathrm{EDHMLO}}_{\sigma,S\uplus\{s\}}(\varrho)$;

– $\mathscr{F}^{\mathrm{EDHMLO}}_{\sigma,S}((@^{J,N}s)\varrho) = (@^{I(\sigma)(J),O(\sigma)(N)}s)\mathscr{F}^{\mathrm{EDHMLO}}_{\sigma,S}(\varrho)$;

– $\mathscr{F}^{\mathrm{EDHMLO}}_{\sigma,S}(\Box^{J,N}\varrho) = \Box^{I(\sigma)(J),O(\sigma)(N)}\mathscr{F}^{\mathrm{EDHMLO}}_{\sigma,S}(\varrho)$;

– $\mathscr{F}^{\mathrm{EDHMLO}}_{\sigma,S}(\Diamond i[O]_N\psi\varrho) =$
$\Diamond I(\sigma)(i)[O(\sigma)(O)]_{O(\sigma)(N)}\mathscr{F}^{2\mathcal{D}}_{A(\sigma),X(i)\cup X(O)}(\psi)\mathscr{F}^{\mathrm{EDHMLO}}_{\sigma,S}(\varrho)$;

– $\mathscr{F}^{\mathrm{EDHMLO}}_{\sigma,S}(\langle\!|i : \phi /\!/ [O]_N : \psi|\!\rangle\varrho) =$
$\langle\!|I(\sigma)(i) : \mathscr{F}^{\mathcal{D}}_{A(\sigma),X(i)}(\phi) /\!/ [O(\sigma)(O)]_{O(\sigma)(N)} : \mathscr{F}^{2\mathcal{D}}_{A(\sigma),X(i)\cup X(O)}(\psi)|\!\rangle\mathscr{F}^{\mathrm{EDHMLO}}_{\sigma,S}(\varrho)$;

– $\mathscr{F}^{\mathrm{EDHMLO}}_{\sigma,S}(\neg\varrho) = \neg\mathscr{F}^{\mathrm{EDHMLO}}_{\sigma,S}(\varrho)$;

– $\mathscr{F}^{\mathrm{EDHMLO}}_{\sigma,S}(\varrho_1 \vee \varrho_2) = \mathscr{F}^{\mathrm{EDHMLO}}_{\sigma,S}(\varrho_1) \vee \mathscr{F}^{\mathrm{EDHMLO}}_{\sigma,S}(\varrho_2)$.

The set $\mathrm{Sen}^{\mathrm{EDHMLO}}(\Sigma)$ of Σ-*event/data sentences* is given by $\mathscr{F}^{\mathrm{EDHMLO}}_{\Sigma,\emptyset}$, the *event/data sentence translation* $\mathrm{Sen}^{\mathrm{EDHMLO}}(\sigma) : \mathrm{Sen}^{\mathrm{EDHMLO}}(\Sigma) \to \mathrm{Sen}^{\mathrm{EDHMLO}}(\Sigma')$ by $\mathscr{F}^{\mathrm{EDHMLO}}_{\sigma,\emptyset}$.

Example 9.4 In EDHMLO, just like in EDHML, we can express that a given class of transition must be possible, or that it must be impossible.

For the ATM machine in Fig. 9.1, we could have a set of state variables

$$S = \{\mathsf{cardId}, \mathsf{pin}, \mathsf{trialsNum}\},$$

the identity function as a state variable assignment v, as well as a data state d given by:

$$\begin{array}{lllr} d & : & \mathsf{cardId} & \mapsto & 7 \\ & & \mathsf{pin} & \mapsto & 42 \\ & & \mathsf{trialsNum} & \mapsto & 2 \end{array}$$

The transition from PINEntered to Verifying could be encoded in the following formula:

$$\begin{aligned} (@\mathsf{PINEntered})\langle\!|\, & \mathsf{complOfATM.PINEntered} : \mathtt{true} \\ & /\!/ [\{\mathsf{bankCom.verify}(\mathsf{cardId}, \mathsf{pin})\}]_\emptyset \\ & : \mathrm{cl}_A(\mathtt{true}) \,|\!\rangle \mathsf{CardEntered} \end{aligned}$$

This claims that in state PINEntered, there is a possible transition, regardless of the data state, which is triggered by the completion event for PINEntered[1], and emitting exactly the event bankCom.verify, which is given as arguments the current values of cardId and pin. No data is changed, and the transition lands in the control state CardEntered.

Likewise, the transition from Verifying to CardEntered could be expressed by:

$$\begin{aligned} (@\mathsf{Verifying})\langle\!|\, & \mathsf{bankCom.reenterPIN} : \mathsf{trialsNum} < 3 \\ & /\!/ [\emptyset]_\emptyset : \mathrm{cl}_A(\mathsf{trialsNum}' = \mathsf{trialsNum} + 1) \\ & |\!\rangle \mathsf{CardEntered} \end{aligned}$$

This asserts that in any configuration with control state Verifying, if trialsNum evaluates to a number strictly less than 3, then there is a transition with no outputs, increasing trialsNum by 1, leaving all other attributes unchanged and landing in control state CardEntered

9.7 Satisfaction Relation for EDHMLO

Let Σ be an event/data signature, M a Σ-event/data structure, S a set of state variables, $v : S \to C(M)$ a state variable assignment, and $\gamma \in \Gamma(M)$. The *satisfaction relation* for event/data formulæ is inductively given by

- $M, v, \gamma \models^{\mathrm{EDHMLO}}_{\Sigma,S} \varphi$ iff $\omega(M)(\gamma), \emptyset \models^{\mathcal{D}}_{A(\Sigma),\emptyset} \varphi$;
- $M, v, \gamma \models^{\mathrm{EDHMLO}}_{\Sigma,S} s$ iff $v(s) = c(M)(\gamma)$;
- $M, v, \gamma \models^{\mathrm{EDHMLO}}_{\Sigma,S} \downarrow s\,.\,\varrho$ iff $M, v\{s \mapsto c(M)(\gamma)\}, \gamma \models^{\mathrm{EDHMLO}}_{\Sigma,S \uplus \{s\}} \varrho$;
- $M, v, \gamma \models^{\mathrm{EDHMLO}}_{\Sigma,S} (@^{J,N} s)\varrho$ iff $M, v, \gamma' \models^{\mathrm{EDHMLO}}_{\Sigma,S} \varrho$ for all $\gamma' \in \Gamma^{J,N}(M)$ with $c(M)(\gamma') = v(s)$;
- $M, v, \gamma \models^{\mathrm{EDHMLO}}_{\Sigma,S} \Box^{J,N}\varrho$ iff $M, v, \gamma' \models^{\mathrm{EDHMLO}}_{\Sigma,S} \varrho$ for all $\gamma' \in \Gamma^{J,N}(M, \gamma)$;

[1] The completion event is delivered over a special internal port for completion events

– $M, v, \gamma \models^{\text{EDHMLO}}_{\Sigma,S} \Diamond i[O]_N \psi \varrho$
iff there are valuations $\beta : X(i) \to \mathcal{D}$, $\beta' : X(O) \to \mathcal{D}$,
output messages $\hat{O}' \in O(\beta') \parallel \hat{N}$ with $\hat{N} \in \hat{O}(N)^*$,
and a configuration $\gamma' \in \Gamma(M)$
such that $(\gamma, \gamma') \in R(M)_{i(\beta),\hat{O}'}$, $(\omega(M)(\gamma), \omega(M)(\gamma')), \beta \cup \beta' \models^{2\mathcal{D}}_{A(\Sigma),X(i)\cup X(O)} \psi$, and $M, v, \gamma' \models^{\text{EDHMLO}}_{\Sigma,S} \varrho$;

– $M, v, \gamma \models^{\text{EDHMLO}}_{\Sigma,S} \langle\!| i : \phi /\!/ [O]_N : \psi |\!\rangle \varrho$
iff for all valuations $\beta : X(i) \to \mathcal{D}$ that satisfy $\omega(M)(\gamma), \beta \models^{\mathcal{D}}_{A(\Sigma),X(i)} \phi$
there are a valuation $\beta' : X(O) \to \mathcal{D}$, output messages $\hat{O}' \in O(\beta') \parallel \hat{N}$ with $\hat{N} \in \hat{O}(N)^*$,
and a configuration $\gamma' \in \Gamma(M)$
such that $(\gamma, \gamma') \in R(M)_{i(\beta),\hat{O}'}$, $(\omega(M)(\gamma), \omega(M)(\gamma')), \beta \cup \beta' \models^{2\mathcal{D}}_{A(\Sigma),X(i)\cup X(O)} \psi$,
and $M, v, \gamma' \models^{\text{EDHMLO}}_{\Sigma,S} \varrho$;

– $M, v, \gamma \models^{\text{EDHMLO}}_{\Sigma,S} \neg\varrho$ iff $M, v, \gamma \not\models^{\text{EDHMLO}}_{\Sigma,S} \varrho$;

– $M, v, \gamma \models^{\text{EDHMLO}}_{\Sigma,S} \varrho_1 \vee \varrho_2$ iff $M, v, \gamma \models^{\text{EDHMLO}}_{\Sigma,S} \varrho_1$ or $M, v, \gamma \models^{\text{EDHMLO}}_{\Sigma,S} \varrho_2$.

For a $\Sigma \in |\text{Sig}^{\text{EDHMLO}}|$, an $M \in |\text{Mod}^{\text{EDHMLO}}(\Sigma)|$, and a $\rho \in \text{Sen}^{\text{EDHMLO}}(\Sigma)$ the *satisfaction relation* $M \models^{\text{EDHMLO}}_{\Sigma} \rho$ holds if, and only if, $M, \emptyset, \gamma_0 \models^{\text{EDHMLO}}_{\Sigma,\emptyset} \rho$ for all $\gamma_0 \in \Gamma_0(M)$.

Example 9.5 For the ATM machine in Fig. 9.1, we could again have the set of state variables

$$S = \{\mathsf{cardId}, \mathsf{pin}, \mathsf{trialsNum}\},$$

the identity function as a state variable assignment v, as well as the data state d given by:

$$\begin{array}{lllr} d & : & \mathsf{cardId} & \mapsto & 7 \\ & & \mathsf{pin} & \mapsto & 42 \\ & & \mathsf{trialsNum} & \mapsto & 2 \end{array}$$

Then, the satisfaction of the transition formula from Verifying to CardEntered in the configuration $\gamma = (\text{Verifying}, d)$ would be expressed as:

$$\begin{aligned} (\Gamma, R, \Gamma_0, \omega), v, \gamma \models^{\text{EDHMLO}}_{\Sigma,S} (@\text{Verifying}) \langle\!| & \mathsf{bankCom.reenterPIN} : \mathsf{trialsNum} < 3 \\ & /\!/ [\emptyset]_\emptyset : \text{cl}_A(\mathsf{trialsNum}' = \mathsf{trialsNum} + 1) \\ & |\!\rangle \text{CardEntered} \end{aligned}$$

This would hold iff $(\gamma, (\text{CardEntered}, d\{\mathsf{trialsNum} \mapsto 3\})) \in R_{\mathsf{bankCom.reenterPIN},\emptyset}$.

9.8 Satisfaction Condition

Theorem 9.6 $(\text{Sig}^{\text{EDHMLO}}, \text{Mod}^{\text{EDHMLO}}, \text{Sen}^{\text{EDHMLO}}, \models^{\text{EDHMLO}})$ is an institution.

Proof. The satisfaction condition for sentences follows directly from that for formulæ, i.e., for signature morphisms $\sigma : \Sigma \to \Sigma'$, models $M' \in |\mathrm{Mod}^{\mathrm{EDHMLO}}(\Sigma')|$, state variables S, state variable assignments $v : S \to C(M')$, configurations $\gamma \in \Gamma(M)$ and formulæ $\mathscr{F}^{\mathrm{EDHMLO}}_{\Sigma,S}$, we have to show that

$$M'|\sigma, v, \gamma \models^{\mathrm{EDHMLO}}_{\Sigma,S} \varrho$$

is equivalent to

$$M', v, \gamma \models^{\mathrm{EDHMLO}}_{\Sigma',S} \mathscr{F}^{\mathrm{EDHMLO}}_{\sigma,S}(\varrho).$$

We prove this by structural induction over ϱ.

Case φ:

$$\begin{aligned}
& M'|\sigma, v, \gamma \models^{\mathrm{EDHMLO}}_{\Sigma,S} \varphi \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\mathrm{EDHMLO}} \} \\
& \omega(M)(\gamma), \emptyset \models^{\mathcal{D}}_{A(\Sigma),\emptyset} \varphi; \\
\Leftrightarrow \quad & \{ \text{ def. } |\sigma \} \\
& \omega(M')(\gamma')|\sigma \models^{\mathcal{D}}_{A(\Sigma),\emptyset} \varphi \\
\Leftrightarrow \quad & \{ \text{ sat. cond. } \mathcal{D} \} \\
& \omega(M')(\gamma') \models^{\mathcal{D}}_{A(\Sigma'),\emptyset} A(\sigma)(\varphi) \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\mathrm{EDHMLO}} \} \\
& M', v, \gamma' \models^{\mathrm{EDHMLO}}_{\Sigma',S} A(\sigma)(\varphi) \\
\Leftrightarrow \quad & \{ \text{ def. } \mathscr{F}^{\mathrm{EDHMLO}}_{\sigma,S} \} \\
& M', v, \gamma \models^{\mathrm{EDHMLO}}_{\Sigma',S} \mathscr{F}^{\mathrm{EDHMLO}}_{\sigma,S}(\varphi)
\end{aligned}$$

Case s:

$$\begin{aligned}
& M'|\sigma, v, \gamma' \models^{\mathrm{EDHMLO}}_{\Sigma,S} s \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\mathrm{EDHMLO}} \} \\
& v(s) = c(M'|\sigma)(\gamma') \\
\Leftrightarrow \quad & \{ \text{ def. } |\sigma \} \\
& v(s) = c(M')(\gamma') \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\mathrm{EDHMLO}} \} \\
& M', v, \gamma' \models^{\mathrm{EDHMLO}}_{\Sigma',S} s \\
\Leftrightarrow \quad & \{ \text{ def. } \mathscr{F}^{\mathrm{EDHMLO}}_{\sigma,S} \} \\
& M', v, \gamma' \models^{\mathrm{EDHMLO}}_{\Sigma',S} \mathscr{F}^{\mathrm{EDHMLO}}_{\sigma,S}(s)
\end{aligned}$$

Case $\downarrow s \,.\, \varrho$:

$$\begin{aligned}
& M'|\sigma, v, \gamma' \models^{\mathrm{EDHMLO}}_{\Sigma,S} \downarrow s \,.\, \varrho \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\mathrm{EDHMLO}} \}
\end{aligned}$$

$$
\begin{aligned}
& M'|\sigma, v\{s \mapsto c(M'|\sigma)(\gamma')\}, \gamma' \models^{\text{EDHMLO}}_{\Sigma, S \uplus \{s\}} \varrho \\
\Leftrightarrow \quad & \{ \text{ def. } |\sigma \text{ and I. H. } \} \\
& M', v\{s \mapsto c(M')(\gamma')\}, \gamma' \models^{\text{EDHMLO}}_{\Sigma', S \uplus \{s\}} \mathscr{F}^{\text{EDHMLO}}_{\sigma, S \uplus \{s\}}(\varrho) \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\text{EDHMLO}} \} \\
& M', v, \gamma' \models^{\text{EDHMLO}}_{\Sigma', S} \downarrow s \,.\, \mathscr{F}^{\text{EDHMLO}}_{\sigma, S \uplus \{s\}}(\varrho) \\
\Leftrightarrow \quad & \{ \text{ def. } \mathscr{F}^{\text{EDHMLO}}_{\sigma, S} \} \\
& M', v, \gamma' \models^{\text{EDHMLO}}_{\Sigma', S} \mathscr{F}^{\text{EDHMLO}}_{\sigma, S}(\downarrow s \,.\, \varrho)
\end{aligned}
$$

Case $(@^{J,N}s)\varrho$:

$$
\begin{aligned}
& M'|\sigma, v, \gamma' \models^{\text{EDHMLO}}_{\Sigma, S} (@^{J,N}s)\varrho \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\text{EDHMLO}} \} \\
& M'|\sigma, v, \gamma'' \models^{\text{EDHMLO}}_{\Sigma, S} \varrho \\
& \quad \text{for all } \gamma'' \in \Gamma^{J,N}(M'|\sigma) \text{ with } c(M'|\sigma)(\gamma'') = v(s) \\
\Leftrightarrow \quad & \{ \text{ def. } |\sigma; \Gamma \text{ reachable from } \Gamma_0 \text{ via } R \} \\
& M'|\sigma, v, \gamma'' \models^{\text{EDHMLO}}_{\Sigma, S} \varrho \\
& \quad \text{for all } \gamma'' \in \Gamma^{I(\sigma)(J), O(\sigma)(N)}(M') \text{ with } c(M'|\sigma)(\gamma'') = v(s) \\
\Leftrightarrow \quad & \{ \text{ I. H. } \} \\
& M', v, \gamma'' \models^{\text{EDHMLO}}_{\Sigma, S} \mathscr{F}^{\text{EDHMLO}}_{\sigma, S}(\varrho) \\
& \quad \text{for all } \gamma'' \in \Gamma^{I(\sigma)(J), O(\sigma)(N)}(M') \text{ with } c(M')(\gamma'') = v(s) \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\text{EDHMLO}} \} \\
& M', v, \gamma' \models^{\text{EDHMLO}}_{\Sigma', S} (@^{I(\sigma)(J), O(\sigma)(N)}s)\mathscr{F}^{\text{EDHMLO}}_{\sigma, S}(\varrho) \\
\Leftrightarrow \quad & \{ \text{ def. } \mathscr{F}^{\text{EDHMLO}}_{\sigma, S} \} \\
& M', v, \gamma' \models^{\text{EDHMLO}}_{\Sigma', S} \mathscr{F}^{\text{EDHMLO}}_{\sigma, S}((@^{J,N}s)\varrho)
\end{aligned}
$$

Case $\Box^{J,N}\varrho$:

$$
\begin{aligned}
& M'|\sigma, v, \gamma' \models^{\text{EDHMLO}}_{\Sigma, S} \Box^{J,N}\varrho \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\text{EDHMLO}} \} \\
& M'|\sigma, v, \gamma'' \models^{\text{EDHMLO}}_{\Sigma, S} \varrho \quad \text{for all } \gamma'' \in \Gamma^{J,N}(M'|\sigma, \gamma') \\
\Leftrightarrow \quad & \{ \text{ def. } |\sigma; \Gamma \text{ reachable from } \Gamma_0 \text{ via } R \} \\
& M'|\sigma, v, \gamma'' \models^{\text{EDHMLO}}_{\Sigma, S} \varrho \quad \text{for all } \gamma'' \in \Gamma^{I(\sigma)(J), O(\sigma)(N)}(M'|\sigma, \gamma') \\
\Leftrightarrow \quad & \{ \text{ I. H. } \} \\
& M', v, \gamma'' \models^{\text{EDHMLO}}_{\Sigma', S} \mathscr{F}^{\text{EDHMLO}}_{\sigma, S}(\varrho) \quad \text{for all } \gamma'' \in \Gamma^{I(\sigma)(J), O(\sigma)(N)}(M', \gamma') \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\text{EDHMLO}} \} \\
& M', v, \gamma' \models^{\text{EDHMLO}}_{\Sigma', S} \Box^{J,N} \mathscr{F}^{\text{EDHMLO}}_{\sigma, S}(\varrho) \\
\Leftrightarrow \quad & \{ \text{ def. } \mathscr{F}^{\text{EDHMLO}}_{\sigma, S} \}
\end{aligned}
$$

$$M', v, \gamma' \models^{\text{EDHMLO}}_{\Sigma', S} \mathscr{F}^{\text{EDHMLO}}_{\sigma, S}(\Box^{J,N}\varrho)$$

Case $\Diamond i[O]_N \psi \varrho$:

$$
\begin{aligned}
& M'|\sigma, v, \gamma' \models^{\text{EDHMLO}}_{\Sigma, S} \Diamond i[O]_N \psi \varrho \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\text{EDHMLO}} \} \\
& M'|\sigma, v, \gamma'' \models^{\text{EDHMLO}}_{\Sigma, S} \varrho \\
& \quad \text{for some } \beta : X(i) \to \mathcal{D}, \\
& \quad \beta' : X(O) \to \mathcal{D}, \\
& \quad \hat{O}' \in O(\beta') \parallel \hat{N} \text{ with } \hat{N} \in \hat{O}(N)^*, \\
& \quad \gamma'' \in \Gamma(M'|\sigma) \\
& \quad \text{and } (\gamma', \gamma'') \in R(M'|\sigma)_{i(\beta), \hat{O}'} \\
& \quad \text{with } (\omega(M'|\sigma)(\gamma'), \omega(M'|\sigma)(\gamma'')), \beta \cup \beta' \models^{2\mathcal{D}}_{A(\Sigma), X(i) \cup X(O)} \psi \\
\Leftrightarrow \quad & \{ \text{ event signature morphism preserve parameter sorts } \} \\
& M'|\sigma, v, \gamma'' \models^{\text{EDHMLO}}_{\Sigma, S} \varrho \\
& \quad \text{for some } \beta : X(I(\sigma)(i)) \to \mathcal{D}, \\
& \quad \beta' : X(O(\sigma)(O)) \to \mathcal{D}, \\
& \quad \hat{O}' \in O(\beta') \parallel \hat{N} \text{ with } \hat{N} \in \hat{O}(N)^*, \\
& \quad \gamma'' \in \Gamma(M'|\sigma) \\
& \quad \text{and } (\gamma', \gamma'') \in R(M'|\sigma)_{i(\beta), \hat{O}'} \\
& \quad \text{with } (\omega(M'|\sigma)(\gamma'), \omega(M'|\sigma)(\gamma'')), \beta \cup \beta' \models^{2\mathcal{D}}_{A(\Sigma), X(i) \cup X(O)} \psi \\
\Leftrightarrow \quad & \{ \text{ def. } |\sigma; \Gamma \text{ reachable from } \Gamma_0 \text{ via } R \} \\
& M'|\sigma, v, \gamma'' \models^{\text{EDHMLO}}_{\Sigma, S} \varrho \\
& \quad \text{for some } \beta : X(I(\sigma)(i)) \to \mathcal{D}, \\
& \quad \beta' : X(O(\sigma)(O)) \to \mathcal{D}, \\
& \quad \hat{O}' \in O(\sigma)(O)(\beta') \parallel \hat{N} \text{ with } \hat{N} \in \hat{O}(N)^*, \\
& \quad \gamma'' \in \Gamma(M'|\sigma) \\
& \quad \text{and } (\gamma', \gamma'') \in R(M')_{I(\sigma)(i)(\beta), \hat{O}'} \\
& \quad \text{with } (\omega(M'|\sigma)(\gamma'), \omega(M'|\sigma)(\gamma'')), \beta \cup \beta' \models^{2\mathcal{D}}_{A(\Sigma), X(i) \cup X(O)} \psi \\
\Leftrightarrow \quad & \{ \text{ sat. cond. } 2\mathcal{D} \} \\
& M'|\sigma, v, \gamma'' \models^{\text{EDHMLO}}_{\Sigma, S} \varrho \\
& \quad \text{for some } \beta : X(I(\sigma)(i)) \to \mathcal{D}, \\
& \quad \beta' : X(O(\sigma)(O)) \to \mathcal{D}, \\
& \quad \hat{O}' \in O(\sigma)(O)(\beta') \parallel \hat{N} \text{ with } \hat{N} \in \hat{O}(N)^*, \\
& \quad \gamma'' \in \Gamma(M'|\sigma) \\
& \quad \text{and } (\gamma', \gamma'') \in R(M')_{I(\sigma)(i)(\beta), \hat{O}'} \\
& \quad \text{with } (\omega(M')(\gamma), \omega(M')(\gamma')), \beta \cup \beta' \models^{2\mathcal{D}}_{A(\Sigma), X(i) \cup X(O)} \mathscr{F}^{2\mathcal{D}}_{A(\sigma), X}(\psi) \\
\Leftrightarrow \quad & \{ \text{ I.H. } \}
\end{aligned}
$$

$$
\begin{aligned}
& M', v, \gamma'' \models^{\text{EDHMLO}}_{\Sigma', S} \Diamond I(\sigma(i)[O(\sigma)(O)]_{O(\sigma)(N)} \psi \mathscr{F}^{\text{EDHMLO}}_{\sigma, S}(\varrho) \\
& \quad \text{for some } \beta : X(I(\sigma(i)) \to \mathcal{D}, \\
& \quad \beta' : X(O(\sigma)(O)) \to \mathcal{D}, \\
& \quad \hat{O}' \in O(\sigma)(O(\beta') \parallel \hat{N} \text{ with } \hat{N} \in \hat{O}(N)^*, \\
& \quad \gamma'' \in \Gamma(M') \\
& \quad \text{and } (\gamma', \gamma'') \in R(M')_{I(\sigma)(i)(\beta), \hat{O}'} \\
& \quad \text{with } (\omega(M'|\sigma)(\gamma'), \omega(M')(\gamma'')), \beta \cup \beta' \models^{2\mathcal{D}}_{A(\Sigma), X(i) \cup X(O)} \psi \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\text{EDHMLO}} \} \\
& M', v, \gamma' \models^{\text{EDHMLO}}_{\Sigma', S} \Diamond I(\sigma(i)[O(\sigma)(O)]_N \psi \mathscr{F}^{\text{EDHMLO}}_{\sigma, S}(\varrho) \\
\Leftrightarrow \quad & \{ \text{ def. } \mathscr{F}^{\text{EDHMLO}}_{\sigma, S} \} \\
& M', v, \gamma' \models^{\text{EDHMLO}}_{\Sigma', S} \mathscr{F}^{\text{EDHMLO}}_{\sigma, S}(\Diamond i[O]_N \psi \varrho)
\end{aligned}
$$

Case $\langle\!| i : \phi /\!/ [O]_N : \psi |\!\rangle \varrho$:

$$
\begin{aligned}
& M'|\sigma, v, \gamma' \models^{\text{EDHMLO}}_{\Sigma, S} \langle\!| i : \phi /\!/ [O]_N : \psi |\!\rangle \varrho \\
\Leftrightarrow \quad & \{ \text{ def. } \models^{\text{EDHMLO}} \} \\
& M'|\sigma, v, \gamma'' \models^{\text{EDHMLO}}_{\Sigma', S} \langle\!| i : \sigma, X(i) /\!/ \phi : [O]_N |\!\rangle \psi \varrho \\
& \quad \text{for all } \beta : X(i) \to \mathcal{D} \text{ satisfying } \omega(M'|\sigma)(\gamma), \beta \models^{\mathcal{D}}_{A(\Sigma), X(i)} \phi \\
& \quad \text{and some } \beta' : X(O) \to \mathcal{D}, \\
& \quad \hat{O}' \in O(\beta') \parallel \hat{N} \text{ with } \hat{N} \in \hat{O}(N)^*, \\
& \quad \gamma'' \in \Gamma(M') \\
& \quad \text{and } (\gamma', \gamma'') \in R(M'|\sigma)_{i(\beta), \hat{O}'} \\
& \quad \text{with } (\omega(M'|\sigma)(\gamma'), \omega(M'|\sigma)(\gamma'')), \beta \cup \beta' \models^{2\mathcal{D}}_{A(\Sigma), X(i) \cup X(O)} \psi \\
\Leftrightarrow \quad & \{ \text{ event signature morphism preserve parameter sorts } \} \\
& M'|\sigma, v, \gamma'' \models^{\text{EDHMLO}}_{\Sigma', S} \langle\!| i : \mathscr{F}^{\mathcal{D}}_{\sigma, A(\sigma), X(i)}(\phi) /\!/ [O(\sigma)(O)]_N : \mathscr{F}^{2\mathcal{D}}_{\sigma, A(\sigma), X(i)}(\psi) |\!\rangle \varrho \\
& \quad \text{for all } \beta : X(I(\sigma(i)) \to \mathcal{D} \text{ satisfying } \omega(M')(\gamma)|A(\sigma), \beta \models^{\mathcal{D}}_{A(\Sigma), X(i)} \phi \\
& \quad \text{and some } \beta' : X(O(\sigma)(O)) \to \mathcal{D}, \\
& \quad \hat{O(\sigma)(O)}' \in O(\sigma)(O(\beta') \parallel \hat{N} \text{ with } \hat{N} \in \hat{O}(N)^*, \\
& \quad \gamma'' \in \Gamma(M') \\
& \quad \text{and } (\gamma', \gamma'') \in R(M'|\sigma)_{i(\beta), \hat{O}'} \\
& \quad \text{with } (\omega(M')(\gamma')|A(\sigma), \omega(M')(\gamma'')|A(\sigma)), \beta \cup \beta' \models^{2\mathcal{D}}_{A(\Sigma), X(i) \cup X(O(\sigma)(O))} \psi \\
\Leftrightarrow \quad & \{ \text{ def. } |\sigma; \Gamma \text{ reachable from } \Gamma_0 \text{ via } R \} \\
& M'|\sigma, v, \gamma'' \models^{\text{EDHMLO}}_{\Sigma', S} \langle\!| i : \mathscr{F}^{\mathcal{D}}_{\sigma, A(\sigma), X(i)}(\phi) /\!/ [O(\sigma)(O)]_N : \mathscr{F}^{2\mathcal{D}}_{\sigma, A(\sigma), X(i)}(\psi) |\!\rangle \varrho \\
& \quad \text{for all } \beta : X(I(\sigma(i)) \to \mathcal{D} \text{ satisfying } \omega(M')(\gamma)|A(\sigma), \beta \models^{\mathcal{D}}_{A(\Sigma), X(i)} \phi \\
& \quad \text{and some } \beta' : X(O(\sigma)(O)) \to \mathcal{D}, \\
& \quad \hat{O(\sigma)(O)}' \in O(\sigma)(O(\beta') \parallel \hat{N} \text{ with } \hat{N} \in \hat{O}(N)^*, \\
& \quad \gamma'' \in \Gamma(M') \\
& \quad \text{and } (\gamma', \gamma'') \in R(M')_{I(\sigma)(i)(\beta), \hat{O(\sigma)(O)}'} \\
& \quad \text{with } (\omega(M')(\gamma')|A(\sigma), \omega(M')(\gamma'')|A(\sigma)), \beta \cup \beta' \models^{2\mathcal{D}}_{A(\Sigma), X(i) \cup X(O(\sigma)(O))} \psi
\end{aligned}
$$

$\Leftrightarrow$ $\{$ sat. cond. $2\mathcal{D}$; sat. cond. $2\mathcal{D}$ $\}$

$M'|\sigma, v, \gamma'' \models^{\text{EDHMLO}}_{\Sigma,S} \langle\!|i : \mathscr{F}^{\mathcal{D}}_{\sigma,A(\sigma),X(i)}(\phi) /\!\!/ [O(\sigma)(O)]_N : \mathscr{F}^{2\mathcal{D}}_{\sigma,A(\sigma),X(i)}(\psi)|\!\rangle \varrho$
for all $\beta : X(I(\sigma(i))) \to \mathcal{D}$ satisfying $\omega(M')(\gamma), \beta \models^{\mathcal{D}}_{A(\Sigma'),X(i)} \mathscr{F}^{\mathcal{D}}_{\sigma,A(\sigma),X(i)}(\phi)$
and some $\beta' : X(O(\sigma)(O)) \to \mathcal{D}$,
$\widehat{O(\sigma)(O)}' \in O(\sigma)(O(\beta') \parallel \hat{N}$ with $\hat{N} \in \hat{O}(N)^*$,
$\gamma'' \in \Gamma(M')$
and $(\gamma', \gamma'') \in R(M')_{I(\sigma)(i)(\beta),\widehat{O(\sigma)(O)}'}$
with $(\omega(M')(\gamma'), \omega(M')(\gamma'')), \beta \cup \beta' \models^{2\mathcal{D}}_{A(\Sigma'),X(i)\cup X(O(\sigma)(O))} \mathscr{F}^{2\mathcal{D}}_{\sigma,A(\sigma),X(i)}(\psi)$

$\Leftrightarrow$ $\{$ I.H. $\}$

$M', v, \gamma'' \models^{\text{EDHMLO}}_{\Sigma',S} \langle\!|i : \mathscr{F}^{\mathcal{D}}_{\sigma,A(\sigma),X(i)}(\phi) /\!\!/ [O(\sigma)(O)]_N : \mathscr{F}^{2\mathcal{D}}_{\sigma,A(\sigma),X(i)}(\psi)|\!\rangle \mathscr{F}^{\text{EDHMLO}}_{\sigma,S}(\varrho)$
for all $\beta : X(I(\sigma(i))) \to \mathcal{D}$ satisfying $\omega(M')(\gamma), \beta \models^{\mathcal{D}}_{A(\Sigma'),X(i)} \mathscr{F}^{\mathcal{D}}_{\sigma,A(\sigma),X(i)}(\phi)$
and some $\beta' : X(O(\sigma)(O)) \to \mathcal{D}$,
$\widehat{O(\sigma)(O)}' \in O(\sigma)(O(\beta') \parallel \hat{N}$ with $\hat{N} \in \hat{O}(N)^*$,
$\gamma'' \in \Gamma(M')$
and $(\gamma', \gamma'') \in R(M')_{I(\sigma)(i)(\beta),\widehat{O(\sigma)(O)}'}$
with $(\omega(M')(\gamma'), \omega(M')(\gamma'')), \beta \cup \beta' \models^{2\mathcal{D}}_{A(\Sigma'),X(i)\cup X(O(\sigma)(O))} \mathscr{F}^{2\mathcal{D}}_{\sigma,A(\sigma),X(i)}(\psi)$

$\Leftrightarrow$ $\{$ def. $\models^{\text{EDHMLO}}$ $\}$

$M', v, \gamma' \models^{\text{EDHMLO}}_{\Sigma',S}$
$\langle\!|I(\sigma)(i) : \mathscr{F}^{\mathcal{D}}_{\sigma,A(\sigma),X(i)}(\phi) /\!\!/ [O(\sigma)(O)]_{N)} : \mathscr{F}^{2\mathcal{D}}_{\sigma,A(\sigma),X(i)}(\psi)|\!\rangle \mathscr{F}^{\text{EDHMLO}}_{\sigma,S}(\varrho)$

$\Leftrightarrow$ $\{$ def. $\mathscr{F}^{\text{EDHMLO}}_{\sigma,S}$ $\}$

$M', v, \gamma' \models^{\text{EDHMLO}}_{\Sigma',S} \mathscr{F}^{\text{EDHMLO}}_{\sigma,S}(\langle\!|i : \phi /\!\!/ [O]_N : \psi|\!\rangle \varrho)$

Case $\neg\varrho$:

$M'|\sigma, v, \gamma' \models^{\text{EDHMLO}}_{\Sigma,S} \neg\varrho$

$\Leftrightarrow$ $\{$ def. $\models^{\text{EDHMLO}}$ $\}$

$M'|\sigma, v, \gamma' \not\models^{\text{EDHMLO}}_{\Sigma,S} \varrho$

$\Leftrightarrow$ $\{$ I.H. $\}$

$M', v, \gamma' \not\models^{\text{EDHMLO}}_{\Sigma',S} \mathscr{F}^{\text{EDHMLO}}_{\sigma,S}(\varrho)$

$\Leftrightarrow$ $\{$ def. $\models^{\text{EDHMLO}}$ $\}$

$M', v, \gamma' \models^{\text{EDHMLO}}_{\Sigma',S} \neg\mathscr{F}^{\text{EDHMLO}}_{\sigma,S}(\varrho)$

$\Leftrightarrow$ $\{$ def. $\mathscr{F}^{\text{EDHMLO}}_{\sigma,S}$ $\}$

$M', v, \gamma' \models^{\text{EDHMLO}}_{\Sigma',S} \mathscr{F}^{\text{EDHMLO}}_{\sigma,S}(\neg\varrho)$

Case $\varrho \vee \varsigma$:

$M'|\sigma, v, \gamma' \models^{\text{EDHMLO}}_{\Sigma,S} \varrho \vee \varsigma$

$\Leftrightarrow$ $\{$ def. $\models^{\text{EDHMLO}}$ $\}$

$M'|\sigma, v, \gamma' \models^{\text{EDHMLO}}_{\Sigma,S} \varrho$ or $M'|\sigma, v, \gamma' \models^{\text{EDHMLO}}_{\Sigma,S} \varsigma$

$\Leftrightarrow \quad \{ \text{I. H.} \}$

$M', v, \gamma' \models^{\text{EDHMLO}}_{\Sigma', S} \mathscr{F}^{\text{EDHMLO}}_{\sigma, S}(\varrho)$ or $M', v, \gamma' \models^{\text{EDHMLO}}_{\Sigma', S} \mathscr{F}^{\text{EDHMLO}}_{\sigma, S}(\varsigma)$

$\Leftrightarrow \quad \{ \text{def. } \models^{\text{EDHMLO}} \}$

$M', v, \gamma' \models^{\text{EDHMLO}}_{\Sigma', S} \mathscr{F}^{\text{EDHMLO}}_{\sigma, S}(\varrho) \vee \mathscr{F}^{\text{EDHMLO}}_{\sigma, S}(\varsigma)$

$\Leftrightarrow \quad \{ \text{def. } \mathscr{F}^{\text{EDHMLO}}_{\sigma, S} \}$

$M', v, \gamma' \models^{\text{EDHMLO}}_{\Sigma', S} \mathscr{F}^{\text{EDHMLO}}_{\sigma, S}(\varrho \vee \varsigma)$

□

Chapter 10

A Theoroidal Comorphism from EDHMLO to CASL

Contents

In this chapter, we define a (theoroidal) comorphism from EDHMLO to CASL, extending the comorphism from EDHML. The construction mainly follows the standard translation of modal logics to first-order logic [VPMY04] and extends the scheme of Chapter 8 by outputs.

The contents of this chapter have been published in similar form as part of [RKR22].

The construction of the comorphism was a common effort between Alexander Knapp, Markus Roggenbach and myself. In our virtual meetings, we jointly sketched the constructions, proofs and examples, which I then worked out in detail and wrote down in their final form.

10.1 Translating Signatures

The basis is a representation of EDHMLO-signatures and the frame given by EDHMLO-structures as a CASL-specification as shown in Fig. 10.1. The signature translation

$$\nu^{\mathrm{Sig}} : \mathrm{Sig}^{\mathrm{EDHMLO}} \to \mathrm{Pres}^{\mathrm{CASL}}$$

maps an EDHMLO-signature Σ to the CASL-theory presentation given by TRANS_Σ and an EDHMLO-signature morphism to the corresponding theory presentation morphism. TRANS_Σ first of all covers the events according to $I(\Sigma)$ and $O(\Sigma)$ with types InEvt and OutEvt, and the configurations with type Conf showing a single constructor conf for the control state from Ctrl and a data state given by assignments to the attributes from $A(\Sigma)$. Furthermore, TRANS_Σ sets the frame for describing reachable transition systems with a set of initial configurations,

from *Basic/StructuredDatatypes* **get** LIST, SET % import finite lists and sets
spec $\text{TRANS}_\Sigma = Dt$
then free type $\text{InEvt} ::= I(\Sigma)$
free type $\text{OutEvt} ::= O(\Sigma)$
then LIST[sort OutEvt] **and** SET[sort InEvt] **and** SET[sort OutEvt]
then sort Ctrl
free type $\text{Conf} ::= \text{conf}(\text{c} : \text{Ctrl}; A(\Sigma))$
preds $\text{init} : \text{Conf}$;
$\text{trans} : \text{Conf} \times \text{InEvt} \times \mathit{List}[\text{OutEvt}] \times \text{Conf}$
$\cdot \exists g : \text{Conf} \cdot \text{init}(g)$ % there is some initial configuration
then free { **pred** $\text{reachable} : \mathit{Set}[\text{InEvt}] \times \mathit{Set}[\text{OutEvt}] \times \text{Conf} \times \text{Conf}$
$\forall g, g', g'' : \text{Conf}; J : \mathit{Set}[\text{InEvt}]; N : \mathit{Set}[\text{OutEvt}]; i : \text{InEvt}; O : \mathit{List}[\text{OutEvt}]$
$\cdot \text{reachable}(J, N, g, g)$
$\cdot \text{reachable}(J, N, g, g') \wedge i \in J \wedge O \subseteq N \wedge \text{trans}(g', i, O, g'') \Rightarrow$
$\text{reachable}(J, N, g, g'')$ }
then **preds** $\text{reachable}(J : \mathit{Set}[\text{InEvt}], N : \mathit{Set}[\text{OutEvt}], g : \text{Conf}) \Leftrightarrow$
$\exists g_0 : \text{Conf} \cdot \text{init}(g_0) \wedge \text{reachable}(J, N, g_0, g)$;
$\text{reachable}(g : \text{Conf}) \Leftrightarrow \text{reachable}(I(\Sigma), O(\Sigma), g)$
then **pred** $\text{mixed} : \mathit{List}[\text{OutEvt}] \times \mathit{Set}[\text{OutEvt}] \times \mathit{List}[\text{OutEvt}]$
$\forall o, o' : \text{OutEvt}; O, O' : \mathit{List}[\text{OutEvt}]; N : \mathit{Set}[\text{OutEvt}]$
$\cdot \text{mixed}(O, N, O)$
$\cdot \text{mixed}(o :: O, N, o :: O')$ **if** $\text{mixed}(O, N, O')$
$\cdot \text{mixed}(o :: O, N, o' :: O')$ **if** $\text{mixed}(o :: O, N, O') \wedge o' \in N$
end

Figure 10.1: Extended frame for translating EDHMLO into CASL.

a transition relation, and reachability predicates, where the specification of reachable uses CASL's "structured free" construct to ensure reachability to be inductively defined. Finally, a predicate mixed is included for representing the shufflings of a list of outputs with some additional output events.

10.2 Reducing Structures

The model translation

$$\nu^{\text{Mod}}_\Sigma : \text{Mod}^{\text{CASL}}(\nu^{\text{Sig}}(\Sigma)) \to \text{Mod}^{\text{EDHMLO}}(\Sigma)$$

then can rely on this encoding. In particular, for a model $M' \in \text{Mod}^{\text{CASL}}(\nu^{\text{Sig}}(\Sigma))$, there are bijective maps $\iota_{M',\text{Conf}} : \text{Conf}^{M'} \cong \text{Ctrl}^{M'} \times \Omega(A(\Sigma))$ for the configurations as well as $\iota_{M',\text{InEvt}} : \text{InEvt}^{M'} \cong \hat{I}(\Sigma)$ and $\iota_{M',\text{OutEvt}} : \text{OutEvt}^{M'} \cong \hat{O}(\Sigma)$ for the messages. Moreover,

$$\text{mixed}^{M'}(\iota^{-1}_{M',\text{OutEvt}}(\hat{O}), \iota^{-1}_{M',\text{OutEvt}}(N), \iota^{-1}_{M',\text{OutEvt}}(\hat{O}'))$$

holds iff $\hat{O}' \in \hat{O} \parallel \hat{N}'$ with $\hat{N}' \in N^*$.

The EDHMLO-structure resulting from a CASL-model M' of TRANS_Σ can thus be defined by

– $\Gamma(\nu^{\mathrm{Mod}}_{\Sigma}(M')) = \iota^{-1}_{M',\mathrm{Conf}}(\{g^{M'} \in \mathrm{Conf}^{M'} \mid \mathrm{reachable}^{M'}(g^{M'})\})$

– $R(\nu^{\mathrm{Mod}}_{\Sigma}(M'))_{\hat{\imath},\hat{O}} = \{(\gamma,\gamma') \in \Gamma(\nu^{\mathrm{Mod}}_{\Sigma}(M')) \times \Gamma(\nu^{\mathrm{Mod}}_{\Sigma}(M')) \mid$
$\mathrm{trans}^{M'}(\iota_{M',\mathrm{Conf}}(\gamma), \iota^{-1}_{M',\mathrm{InEvt}}(\hat{\imath}), \iota^{-1}_{M',\mathrm{OutEvt}}(\hat{O}), \iota_{M',\mathrm{Conf}}(\gamma'))\}$

– $\Gamma_0(\nu^{\mathrm{Mod}}_{\Sigma}(M')) = \{\gamma \in \Gamma(\nu^{\mathrm{Mod}}_{\Sigma}(M')) \mid \mathrm{init}^{M'}(\iota_{M',\mathrm{Conf}}(\gamma))\})$

– $\omega(\nu^{\mathrm{Mod}}_{\Sigma}(M')) = \{(c,\omega) \in \Gamma(\nu^{\mathrm{Mod}}_{\Sigma}(M')) \mapsto \omega\}$

10.3 Translating Sentences

For EDHMLO-sentences, we first define a formula translation

$$\nu^{\mathscr{F}}_{\Sigma,S,g} : \mathscr{F}^{\mathrm{EDHMLO}}_{\Sigma,S} \to \mathscr{F}^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma),S\cup\{g\}}$$

which, mimicking the standard translation, takes a variable $g : \mathrm{Conf}$ as a parameter that records the "current configuration" and also uses a set S of state names for the control states. The translation embeds the data state and 2-data state formulæ using the substitution $A(\Sigma)(g) = \{a \mapsto a(g) \mid a \in A(\Sigma)\}$ for replacing the attributes $a \in A(\Sigma)$ by the accessors $a(g)$. The translation of EDHMLO-formulæ then reads

– $\nu^{\mathscr{F}}_{\Sigma,S,g}(\varphi) = \mathscr{F}^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma),A(\Sigma)(g)}(\varphi)$

– $\nu^{\mathscr{F}}_{\Sigma,S,g}(s) = (s = \mathrm{c}(g))$

– $\nu^{\mathscr{F}}_{\Sigma,S,g}(\downarrow s\,.\,\varrho) = \exists s : \mathrm{Ctrl}\,.\, s = \mathrm{c}(g) \wedge \nu^{\mathscr{F}}_{\Sigma,S\uplus\{s\},g}(\varrho)$

– $\nu^{\mathscr{F}}_{\Sigma,S,g}((@^{J,N}s)\varrho) = \forall g' : \mathrm{Conf}\,.\,(\mathrm{c}(g') = s \wedge \mathrm{reachable}(J,N,g')) \Rightarrow \nu^{\mathscr{F}}_{\Sigma,S,g'}(\varrho)$

– $\nu^{\mathscr{F}}_{\Sigma,S,g}(\Box^{J,N}\varrho) = \forall g' : \mathrm{Conf}\,.\,\mathrm{reachable}(J,N,g,g') \Rightarrow \nu^{\mathscr{F}}_{\Sigma,S,g'}(\varrho)$

– $\nu^{\mathscr{F}}_{\Sigma,S,g}(\Diamond i[O]_N\psi\varrho) = \exists X : \overline{s}(I(\Sigma))(i); X' : \overline{s}(O(\Sigma))(O);$
$O' : \mathit{List}[\mathrm{OutEvt}]; g' : \mathrm{Conf}\,.$
$\mathrm{mixed}(O(X'),N,O') \wedge \mathrm{trans}(g,i(X),O',g') \wedge$
$\mathscr{F}^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma),A(\Sigma)(g)\cup A(\Sigma)(g')\cup 1_{X\cup X'}}(\psi) \wedge \nu^{\mathscr{F}}_{\Sigma,S,g'}(\varrho)$

– $\nu^{\mathscr{F}}_{\Sigma,S,g}(\langle\!|\, i : \phi /\!/ [O]_N : \psi \,|\!\rangle\varrho) = \forall X : \overline{s}(I(\Sigma))(i)\,.\,\mathscr{F}^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma),A(\Sigma)(g)\cup 1_X}(\phi) \Rightarrow$
$\exists X' : \overline{s}(O(\Sigma))(O); O' : \mathit{List}[\mathrm{OutEvt}]; g' : \mathrm{Conf}\,.$
$\mathrm{mixed}(O(X'),N,O') \wedge \mathrm{trans}(g,i(X),O',g') \wedge$
$\mathscr{F}^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma),A(\Sigma)(g)\cup A(\Sigma)(g')\cup 1_{X\cup X'}}(\psi) \wedge \nu^{\mathscr{F}}_{\Sigma,S,g'}(\varrho)$

– $\nu^{\mathscr{F}}_{\Sigma,S,g}(\neg\varrho) = \neg\nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho)$

– $\nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho_1 \vee \varrho_2) = \nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho_1) \vee \nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho_2)$

Building on the translation of formulæ, the sentence translation

$$\nu^{\mathrm{Sen}}_{\Sigma} : \mathrm{Sen}^{\mathrm{EDHMLO}}(\Sigma) \to \mathrm{Sen}^{\mathrm{CASL}}(\nu^{\mathrm{Sig}}(\Sigma))$$

only has to require additionally that evaluation starts in an initial state:

– $\nu^{\mathrm{Sen}}_{\Sigma}(\rho) = \forall g : \mathrm{Conf}.\mathrm{init}(g) \Rightarrow \nu^{\mathscr{F}}_{\Sigma,\emptyset,g}(\rho)$

10.4 Satisfaction Condition

Lemma 10.1 For a $\varrho \in \mathscr{F}^{\mathrm{EDHMLO}}_{\Sigma,S}$, an $M' \in \mathrm{Mod}^{\mathrm{CASL}}(\nu^{\mathrm{Sig}}(\Sigma))$, a $v : S \to C(\nu^{\mathrm{Mod}}_{\Sigma}(M'))$, and a $\gamma \in \Gamma(\nu^{\mathrm{Mod}}_{\Sigma}(M'))$ it holds with $\beta'_{M',g}(v,\gamma) = \iota^{-1}_{M',\mathrm{Ctrl}} \circ v \cup \{g \mapsto \iota_{M',\mathrm{Conf}}(\gamma)\}$ that

$$\nu^{\mathrm{Mod}}_{\Sigma}(M'), v, \gamma \models^{\mathrm{EDHMLO}}_{\Sigma,S} \varrho \iff M', \beta'_{M',g}(v,\gamma) \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma),S\cup\{g\}} \nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho)\ .$$

Proof. We proceed by structural induction over ϱ. *Case* φ:

$$\begin{array}{ll}
& \nu^{\mathrm{Mod}}_{\Sigma}(M'), v, \gamma \models^{\mathrm{EDHMLO}}_{\Sigma,S} \varphi \\
\Leftrightarrow & \{\ \text{def. } \models^{\mathrm{EDHMLO}}\ \} \\
& \omega(\nu^{\mathrm{Mod}}_{\Sigma}(M'))(\gamma) \models^{\mathcal{D}}_{A(\Sigma),\emptyset} \varphi \\
\Leftrightarrow & \{\ \text{def. } \models^{\mathcal{D}}, \text{def. } \models^{\mathrm{CASL}}\ \} \\
& M', \iota^{-1}_{M',dt} \circ \omega(\nu^{\mathrm{Mod}}_{\Sigma}(M'))(\gamma) \cup \\
& \quad \{g \mapsto \iota_{M',\mathrm{Conf}}(\gamma)\} \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma),\{g\}} \mathscr{F}^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma),A(\Sigma)(g)}(\varphi) \\
\Leftrightarrow & \{\ \text{def. } \models^{\mathrm{CASL}}\ \} \\
& M', \beta'_{M',g}(v,\gamma) \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma),S\cup\{g\}} \mathscr{F}^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma),A(\Sigma)(g)}(\varphi) \\
\Leftrightarrow & \{\ \text{def. } \nu^{\mathscr{F}}\ \} \\
& M', \beta'_{M',g}(v,\gamma) \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma),S\cup\{g\}} \nu^{\mathscr{F}}_{\Sigma,S,g}(\varphi)
\end{array}$$

Case s:

$$\begin{array}{ll}
& \nu^{\mathrm{Mod}}_{\Sigma}(M'), v, \gamma \models^{\mathrm{EDHMLO}}_{\Sigma,S} s \\
\Leftrightarrow & \{\ \text{def. } \models^{\mathrm{EDHMLO}}\ \} \\
& v(s) = c(\nu^{\mathrm{Mod}}_{\Sigma}(M'))(\gamma) \\
\Leftrightarrow & \{\ \text{def. } \nu^{\mathrm{Mod}}, \models^{\mathrm{CASL}}\ \} \\
& M', \beta'_{M',g}(v,\gamma) \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma),S\cup\{g\}} s = \mathsf{c}(g) \\
\Leftrightarrow & \{\ \text{def. } \nu^{\mathscr{F}}\ \} \\
& M', \beta'_{M',g}(v,\gamma) \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma),S\cup\{g\}} \nu^{\mathscr{F}}_{\Sigma,S,g}(s)
\end{array}$$

Case $\downarrow s.\varrho$:

$$\begin{array}{ll}
& \nu^{\mathrm{Mod}}_{\Sigma}(M'), v, \gamma \models^{\mathrm{EDHMLO}}_{\Sigma,S} \downarrow s.\varrho \\
\Leftrightarrow & \{\ \text{def. } \models^{\mathrm{EDHMLO}}\ \}
\end{array}$$

$$
\begin{aligned}
& \nu^{\mathrm{Mod}}_{\Sigma}(M'), v\{s \mapsto c(\nu^{\mathrm{Mod}}_{\Sigma}(M'))(\gamma)\}, \gamma \models^{\mathrm{EDHMLO}}_{\Sigma, S \uplus \{s\}} \varrho \\
\Leftrightarrow \quad & \{\text{ I. H. }\} \\
& M', \beta'_{M',g}(v\{s \mapsto c(\nu^{\mathrm{Mod}}_{\Sigma}(M'))(\gamma)\}, \gamma) \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma), (S \uplus \{s\}) \cup \{g\}} \nu^{\mathscr{F}}_{\Sigma, S \uplus \{s\}, g}(\varrho) \\
\Leftrightarrow \quad & \{\text{ def. } \models^{\mathrm{CASL}} \} \\
& M', \beta'_{M',g}(v, \gamma) \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma), S \cup \{g\}} \exists s : \mathrm{Ctrl} \,.\, s = \mathrm{c}(g) \wedge \nu^{\mathscr{F}}_{\Sigma, S \uplus \{s\}, g}(\varrho) \\
\Leftrightarrow \quad & \{\text{ def. } \nu^{\mathscr{F}} \} \\
& M', \beta'_{M',g}(v, \gamma) \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma), S \cup \{g\}} \nu^{\mathscr{F}}_{\Sigma, S, g}(\downarrow s \,.\, \varrho)
\end{aligned}
$$

Case $(@^{J,N}s)\varrho$:

$$
\begin{aligned}
& \nu^{\mathrm{Mod}}_{\Sigma}(M'), v, \gamma \models^{\mathrm{EDHMLO}}_{\Sigma, S} (@^{J,N}s)\varrho \\
\Leftrightarrow \quad & \{\text{ def. } \models^{\mathrm{EDHMLO}} \} \\
& \nu^{\mathrm{Mod}}_{\Sigma}(M'), v, \gamma' \models^{\mathrm{EDHMLO}}_{\Sigma, S} \varrho \\
& \qquad \text{for all } \gamma' \in \Gamma(\nu^{\mathrm{Mod}}_{\Sigma}(M')) \text{ with } c(\nu^{\mathrm{Mod}}_{\Sigma}(M'))(\gamma') = v(s) \\
\Leftrightarrow \quad & \{\text{ I. H. }\} \\
& M', \beta'_{M',g'}(v, \gamma') \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma), S \cup \{g'\}} \nu^{\mathscr{F}}_{\Sigma, S, g'}(\varrho) \\
& \qquad \text{for all } \gamma' \in \Gamma(\nu^{\mathrm{Mod}}_{\Sigma}(M')) \text{ with } c(\nu^{\mathrm{Mod}}_{\Sigma}(M'))(\gamma') = v(s) \\
\Leftrightarrow \quad & \{\text{ def. } \models^{\mathrm{CASL}} \} \\
& M', \beta'_{M',g}(v, \gamma) \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma), S \cup \{g\}} \\
& \qquad \forall g' : \mathrm{Conf} \,.\, (\mathrm{c}(g') = s \wedge \mathrm{reachable}(J, N, g')) \Rightarrow \nu^{\mathscr{F}}_{\Sigma, S, g'}(\varrho) \\
\Leftrightarrow \quad & \{\text{ def. } \nu^{\mathscr{F}} \} \\
& M', \beta'_{M',g}(v, \gamma) \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma), S \cup \{g\}} \nu^{\mathscr{F}}_{\Sigma, S, g}((@^{J,N}s)\varrho)
\end{aligned}
$$

Case $\Box^{J,N}\varrho$:

$$
\begin{aligned}
& \nu^{\mathrm{Mod}}_{\Sigma}(M'), v, \gamma \models^{\mathrm{EDHMLO}}_{\Sigma, S} \Box^{J,N}\varrho \\
\Leftrightarrow \quad & \{\text{ def. } \models^{\mathrm{EDHMLO}} \} \\
& \nu^{\mathrm{Mod}}_{\Sigma}(M'), v, \gamma \models^{\mathrm{EDHMLO}}_{\Sigma, S} \varrho \quad \text{for all } \gamma' \in \Gamma^{J,N}(M', \gamma) \\
\Leftrightarrow \quad & \{\text{ I. H. }\} \\
& M', \beta'_{M',g'}(v, \gamma') \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma), S \cup \{g'\}} \nu^{\mathscr{F}}_{\Sigma, S, g'}(\varrho) \quad \text{for all } \gamma' \in \Gamma(\nu^{\mathrm{Mod}}_{\Sigma}(M'), \gamma) \\
\Leftrightarrow \quad & \{\text{ def. } \models^{\mathrm{CASL}} \} \\
& M', \beta'_{M',g}(v, \gamma) \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma), S \cup \{g\}} \forall g' : \mathrm{Conf} \,.\, (c(g') = s \wedge \mathrm{reachable}(J, N, g')) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow \nu^{\mathscr{F}}_{\Sigma, S, g'}(\Box^{J,N}\varrho) \\
\Leftrightarrow \quad & \{\text{ def. } \nu^{\mathscr{F}} \} \\
& M', \beta'_{M',g}(v, \gamma) \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma), S \cup \{g\}} \nu^{\mathscr{F}}_{\Sigma, S, g}(\Box^{J,N}\varrho)
\end{aligned}
$$

Case $\Diamond i[O]_N \psi \varrho$:

$\nu^{\mathrm{Mod}}_\Sigma(M'), v, \gamma \models^{\mathrm{EDHMLO}}_{\Sigma,S} \Diamond i[O]_N \psi \varrho$

$\Leftrightarrow$ { def. $\models^{\mathrm{EDHMLO}}$ }

$\nu^{\mathrm{Mod}}_\Sigma(M'), v, \gamma' \models^{\mathrm{EDHMLO}}_{\Sigma,S} \varrho$
for some $\beta_i : X(i) \to \mathcal{D}$,
$\beta_O : X(O) \to \mathcal{D}$,
$\hat{O}' \in O(\beta_O) \parallel \hat{N}$ with $\hat{N} \in \hat{O}(N)^*$,
$\gamma' \in \Gamma(\nu^{\mathrm{Mod}}_\Sigma(M'))$
such that $(\gamma, \gamma') \in R(\nu^{\mathrm{Mod}}_\Sigma(M'))_{i(\beta_i),\hat{O}'}$,
and $(\omega(\nu^{\mathrm{Mod}}_\Sigma(M'))(\gamma), \omega(\nu^{\mathrm{Mod}}_\Sigma(M'))(\gamma')), \beta_i \cup \beta_O \models^{2\mathcal{D}}_{A(\Sigma),X(i)\cup X(O)} \psi$

$\Leftrightarrow$ { I. H., def. $\models^{2\mathcal{D}}$, def. $\models^{\mathrm{CASL}}$ }

$M', \beta'_{M',g}(v, \gamma) \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma),S\cup\{g\}} \nu^{\mathscr{F}}_{\Sigma,S,g'}(\varrho)$
for some $\beta_i : X(i) \to \mathcal{D}$,
$\beta_O : X(O) \to \mathcal{D}$
$\hat{O}' \in O(\beta_O) \parallel \hat{N}$ with $\hat{N} \in \hat{O}(N)^*$,
$\gamma' \in \Gamma(\nu^{\mathrm{Mod}}_\Sigma(M'))$
with $(\gamma, \gamma') \in R(\nu^{\mathrm{Mod}}_\Sigma(M'))_{i(\beta_i),\hat{O}'}$
and
$M', ((\iota^{-1}_{M',\bar{s}(I(\Sigma))(O)} \circ \omega(\nu^{\mathrm{Mod}}_\Sigma(M'))(\gamma) + \circ \iota^{-1}_{M',\bar{s}(O(\Sigma))(O)} \omega(\nu^{\mathrm{Mod}}_\Sigma(M'))(\gamma')) \cup \beta) \cup$
$\{g \mapsto \iota_{M',\mathrm{Conf}}(\gamma), g' \mapsto \iota_{M',\mathrm{Conf}}(\gamma')\} \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma),X\cup\{g,g'\}}$
$\mathscr{F}^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma),A(\Sigma)(g)\cup A(\Sigma)(g')\cup 1_{X\cup X'}}(\psi)$

$\Leftrightarrow$ { def. $\models^{\mathrm{CASL}}$; def. $\nu^{\mathrm{Mod}}_\Sigma(M')$ }

$M', \beta'_{M',g}(v, \gamma) \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma),S\cup\{g,g'\}}$
$\exists X : \bar{s}(I(\Sigma))(i); X' : \bar{s}(O(\Sigma))(O);$
$O' : \mathit{List}[\mathrm{OutEvt}]; g' : \mathrm{Conf}.$
$\mathrm{mixed}(O(X'), N, O') \wedge \mathrm{trans}(g, i(X), O', g') \wedge \nu^{\mathscr{F}}_{\Sigma,S,g'}(\varrho)$

$\Leftrightarrow$ { def. $\nu^{\mathscr{F}}$ }

$M', \beta'_{M',g}(v, \gamma) \models^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma),S\cup\{g\}} \nu^{\mathscr{F}}_{\Sigma,S,g}(\Diamond i[O]_N \psi \varrho)$

Case $\langle\!| i : \phi /\!/ [O]_N : \psi |\!\rangle \varrho$:

$\nu^{\mathrm{Mod}}_\Sigma(M'), v, \gamma \models^{\mathrm{EDHMLO}}_{\Sigma,S} \langle\!| i : \phi /\!/ [O]_N : \psi |\!\rangle \varrho$

$\Leftrightarrow$ { def. $\models^{\mathrm{EDHMLO}}$ }

$\nu^{\mathrm{Mod}}_\Sigma(M'), v, \gamma \models^{\mathrm{EDHMLO}}_{\Sigma,S} \varrho$

for all $\beta_i : X(i) \to \mathcal{D}$ with $\mathscr{F}^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),A(\Sigma)(g)\cup 1_X}(\phi)$
and some $\beta_O : X(O) \to \mathcal{D}$
$\hat{O}' \in O(\beta_O) \parallel \hat{N}$ with $\hat{N} \in \hat{O}(N)^*$,
$\gamma' \in \Gamma(\nu^{\text{Mod}}_\Sigma(M'))$
with $(\gamma,\gamma') \in R(\nu^{\text{Mod}}_\Sigma(M'))_{i(\beta_i),\hat{O}'}$
and $(\omega(\nu^{\text{Mod}}_\Sigma(M'))(\gamma),\omega(\nu^{\text{Mod}}_\Sigma(M'))(\gamma')),\beta_i \cup \beta_O \models^{2\mathcal{D}}_{A(\Sigma),X(i)\cup X(O)} \psi$

$\Leftrightarrow$ { I. H., def. $\models^{2\mathcal{D}}$, def. $\models^{\text{CASL}}$ }

$M',\beta'_{M',g}(v,\gamma) \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),S\cup\{g\}} \nu^{\mathscr{F}}_{\Sigma,S,g'}(\varrho)$
for all $\beta_i : X(i) \to \mathcal{D}$ with $\mathscr{F}^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),A(\Sigma)(g)\cup 1_X}(\phi)$
and some $\beta_O : X(O) \to \mathcal{D}$, $\gamma' \in \Gamma(\nu^{\text{Mod}}_\Sigma(M'))$,
$\hat{O}' \in O(\beta_O) \parallel \hat{N}$ with $\hat{N} \in \hat{O}(N)^*$,
with $(\gamma,\gamma') \in R(\nu^{\text{Mod}}_\Sigma(M'))_{i(\beta_i),\hat{O}'}$
and $M',((\iota^{-1}_{M',\bar{s}(I(\Sigma))(O)} \circ \omega(\nu^{\text{Mod}}_\Sigma(M'))(\gamma) + \circ\iota^{-1}_{M',\bar{s}(O(\Sigma))(O)}\omega(\nu^{\text{Mod}}_\Sigma(M'))(\gamma')) \cup \beta) \cup$
$\{g \mapsto \iota_{M',\text{Conf}}(\gamma), g' \mapsto \iota_{M',\text{Conf}}(\gamma')\} \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),X\cup\{g,g'\}}$
$\mathscr{F}^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),A(\Sigma)(g)\cup A(\Sigma)(g')\cup 1_{X\cup X'}}(\psi)$

$\Leftrightarrow$ { def. $\models^{\text{CASL}}$; def. $\nu^{\text{Mod}}_\Sigma(M')$ }

$\forall X : \bar{s}(I(\Sigma))(i) . \mathscr{F}^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),A(\Sigma)(g)\cup 1_X}(\phi) \Rightarrow$
$\exists X' : \bar{s}(O(\Sigma))(O); O' : \mathit{List}[\text{OutEvt}]; g' : \text{Conf} .$
$\text{mixed}(O(X'),N,O') \wedge \text{trans}(g,i(X),O',g') \wedge$
$\mathscr{F}^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),A(\Sigma)(g)\cup A(\Sigma)(g')\cup 1_{X\cup X'}}(\psi) \wedge \nu^{\mathscr{F}}_{\Sigma,S,g'}(\varrho)$

$\Leftrightarrow$ { def. $\nu^{\mathscr{F}}$ }

$M',\beta'_{M',g}(v,\gamma) \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),S\cup\{g\}} \nu^{\mathscr{F}}_{\Sigma,S,g}(\langle\!| i : \phi /\!/ [O]_N : \psi |\!\rangle \varrho)$

Case $\neg\varrho$:

$\nu^{\text{Mod}}_\Sigma(M'),v,\gamma \models^{\text{EDHMLO}}_{\Sigma,S} \neg\varrho$

$\Leftrightarrow$ { def. $\models^{\text{EDHMLO}}$ }

$\nu^{\text{Mod}}_\Sigma(M'),v,\gamma \not\models^{\text{EDHMLO}}_{\Sigma,S} \varrho$

$\Leftrightarrow$ { I. H. }

$M',\beta'_{M',g}(v,\gamma) \not\models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),S\cup\{g\}} \nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho)$

$\Leftrightarrow$ { def. $\models^{\text{CASL}}$ }

$M',\beta'_{M',g}(v,\gamma) \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),S\cup\{g\}} \neg(\nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho))$

$\Leftrightarrow$ { def. $\nu^{\mathscr{F}}$ }

$M',\beta'_{M',g}(v,\gamma) \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma),S\cup\{g\}} \nu^{\mathscr{F}}_{\Sigma,S,g}(\neg\varrho)$

Case $\varrho_1 \vee \varrho_2$:

$\nu^{\text{Mod}}_\Sigma(M'),v,\gamma \models^{\text{EDHMLO}}_{\Sigma,S} \varrho_1 \vee \varrho_2$

$$
\begin{array}{ll}
\Leftrightarrow & \{ \text{ def. } \models^{\text{EDHMLO}} \} \\
& \nu^{\text{Mod}}_{\Sigma}(M'), v, \gamma \models^{\text{EDHMLO}}_{\Sigma,S} \varrho_1 \text{ or } \nu^{\text{Mod}}_{\Sigma}(M'), v, \gamma \models^{\text{EDHMLO}}_{\Sigma,S} \varrho_2 \\
\Leftrightarrow & \{ \text{ I.H. } \} \\
& M', v, \gamma \models^{\text{EDHMLO}}_{\Sigma,S} \nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho_1) \text{ or } M', v, \gamma \models^{\text{EDHMLO}}_{\Sigma,S} \nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho_2) \\
\Leftrightarrow & \{ \text{ def. } \models^{\text{CASL}} \} \\
& M', \beta'_{M',g}(v, \gamma) \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma), S \cup \{g\}} (\nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho_1)) \vee (\nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho_2)) \\
\Leftrightarrow & \{ \text{ def. } \nu^{\mathscr{F}} \} \\
& M', \beta'_{M',g}(v, \gamma) \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma), S \cup \{g\}} \nu^{\mathscr{F}}_{\Sigma,S,g}(\varrho_1 \vee \varrho_2)
\end{array}
$$

□

Theorem 10.2 $(\nu^{\text{Sig}}, \nu^{\text{Mod}}, \nu^{\text{Sen}})$ is a theoroidal comorphism from EDHMLO to CASL.

Proof. Let $\Sigma \in \text{Sig}^{\text{EDHMLO}}$, $M' \in |\text{Mod}^{\text{CASL}}(\nu^{\text{Sig}}(\Sigma))|$, and $\rho \in \text{Sen}^{\text{EDHMLO}}(\Sigma)$. The satisfaction condition follows from

$$
\begin{array}{ll}
& \nu^{\text{Mod}}_{\Sigma}(M') \models^{\text{EDHMLO}}_{\Sigma} \rho \\
\Leftrightarrow & \{ \text{ def. } \models^{\text{EDHMLO}} \} \\
& \nu^{\text{Mod}}_{\Sigma}(M'), \emptyset, \gamma_0 \models^{\text{EDHMLO}}_{\Sigma,\emptyset} \rho \quad \text{for all } \gamma_0 \in \Gamma_0(\nu^{\text{Mod}}_{\Sigma}(M')) \\
\Leftrightarrow & \{ \text{ Lem. 10.1 } \} \\
& M', \beta'_{M',g}(\emptyset, \gamma_0) \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma), \{g\}} \nu^{\mathscr{F}}_{\Sigma,\emptyset,g}(\rho) \quad \text{for all } \gamma_0 \in \Gamma_0(\nu^{\text{Mod}}_{\Sigma}(M')) \\
\Leftrightarrow & \{ \text{ def. } \models^{\text{CASL}} \} \\
& M' \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma)} \forall g : \text{Conf}.\text{init}(g) \Rightarrow \nu^{\mathscr{F}}_{\Sigma,\emptyset,g}(\rho) \\
\Leftrightarrow & \{ \text{ def. } \nu^{\text{Sen}} \} \\
& M' \models^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma)} \nu^{\text{Sen}}_{\Sigma}(\rho)
\end{array}
$$

□

For a CASL-proof of an EDHMLO-*invariant* $\Box\varphi$ such that φ has to hold in every reachable configuration, the full generality of the reachable predicate can sometimes be avoided by replacing the proof obligation $\forall g : \text{Conf}.\text{reachable}(g) \Rightarrow \mathscr{F}^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma), A(\Sigma)(g)}(\varphi)$ by the usual stepwise induction scheme that only requires to demonstrate the invariant to hold in all initial configurations and that it is preserved by every transition. Moreover, the EDHMLO-state formula φ can be generalised into a CASL-invariant.

Proposition 10.3 Let (Σ, P) be a theory presentation in EDHMLO and $(\nu^{\text{Sig}}(\Sigma), \Phi)$ a theory presentation in CASL such that $\text{Mod}^{\text{CASL}}(\nu^{\text{Pres}}(\Sigma, P)) \subseteq \text{Mod}^{\text{CASL}}(\nu^{\text{Sig}}(\Sigma), \Phi)$. Let $inv^{\text{CASL}}(g) \in \mathscr{F}^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma), \{g\}}$ be a CASL-formula with a single free variable g and $inv^{\text{EDHMLO}} \in \mathscr{F}^{\mathcal{D}}_{A(\Sigma),\emptyset}$ an EDHMLO-state formula, such that

$$\forall g : \text{Conf}.inv^{\text{CASL}}(g) \Rightarrow \mathscr{F}^{\text{CASL}}_{\nu^{\text{Sig}}(\Sigma), A(\Sigma)(g)}(inv^{\text{EDHMLO}}) \tag{I0}$$

$$\forall g : \mathrm{Conf}.\mathrm{init}(g) \Rightarrow inv^{\mathrm{CASL}}(g) \tag{I1}$$

$$\forall g, g' : \mathrm{Conf}; i \in \mathrm{InEvt}; O \in List[\mathrm{OutEvt}]. \quad inv^{\mathrm{CASL}}(g) \wedge \mathrm{trans}(g, i, O, g') \Rightarrow inv^{\mathrm{CASL}}(g') \tag{I2}$$

hold in every model $M' \in \mathrm{Mod}^{\mathrm{CASL}}(\nu^{\mathrm{Sig}}(\Sigma), \Phi)$. Then $\nu_{\Sigma}^{\mathrm{Mod}}(M') \models_{\Sigma}^{\mathrm{EDHMLO}} \Box inv^{\mathrm{EDHMLO}}$ holds for all models $M' \in \mathrm{Mod}^{\mathrm{CASL}}(\nu^{\mathrm{Pres}}(\Sigma, P))$.

Proof. Fix an arbitrary signature Σ and model $M' \in \mathrm{Mod}^{\mathrm{CASL}}(\nu^{\mathrm{Pres}}(\Sigma, P))$. From the assumptions and the definition of reachable we get that in all reachable configurations $\gamma \in M'(\mathrm{reachable})$, the condition $M', \{g := \gamma\} \models^{\mathrm{CASL}} inv^{\mathrm{CASL}}(g)$ holds.

From this, we get

$$M' \models^{\mathrm{CASL}} \forall g : \mathrm{Conf.reachable}(g) \Rightarrow inv^{\mathrm{CASL}}(g),$$

hence

$$M' \models^{\mathrm{EDHMLO}} \forall g : \mathrm{Conf.reachable}(g) \Rightarrow inv^{\mathrm{CASL}}(g),$$

hence

$$M' \models^{\mathrm{EDHMLO}} \mathscr{F}^{\mathrm{CASL}}_{\nu^{\mathrm{Sig}}(\Sigma), A(\Sigma)(g)}(inv^{\mathrm{EDHMLO}})$$

hence, by the satisfaction condition,

$$\nu_{\Sigma}^{\mathrm{Mod}}(M') \models_{\Sigma}^{\mathrm{EDHMLO}} \Box inv^{\mathrm{EDHMLO}}.$$

□

Chapter 11

UMLCOMP: Simple UML Composite Structures

Contents

In this chapter, we present a notion of simple UML Composite Structures built on EDHMLO.

The contents of this chapter have been published in similar form as part of [RKR22].

They result from a common effort between Alexander Knapp, Markus Roggenbach and myself. In our virtual meetings, we jointly sketched the constructions, proofs and examples, which I then worked out in detail and wrote down in their final form.

11.1 Signatures

A UML Composite Structure specifies the internal structure and external connections of a class or component. For our purposes, a Composite Structure is given by class or component instances, its so-called *parts*, that can communicate through their attached *ports* specifying input and output interfaces and being linked by *connectors*. All connectors are assumed to be binary and each part to be equipped with a State Machine for describing its behaviour.

A *Composite Structure signature* Δ over EDHMLO consists of

- a set $Cmp(\Delta)$ of *parts* c,
  - each equipped with an EDHMLO-signature $\mathrm{Sig}(\Delta, c)$ for its input and output events and internal attributes;
- a set $Prt(\Delta)$ of *ports* p, each showing

  - an owning part $cmp(\Delta)(p) \in Cmp(\Delta)$ as well as
  - an EDHMLO-signature $\mathrm{Sig}(\Delta, p)$ without attributes (i.e., $A(\mathrm{Sig}(\Delta, p)) = \emptyset$) for its input and output events; and

- a symmetric binary relation $Con(\Delta) \subseteq Prt(\Delta) \times Prt(\Delta)$ of *connectors*

such that

- for each part $c \in Cmp(\Delta)$, the input and output events of $\mathrm{Sig}(\Delta, c)$ are the input and output events of c's ports prefixed with the port name, i.e., for $F \in \{I, O\}$, $F(\mathrm{Sig}(\Delta, c)) = \bigcup_{p \in cmp(\Delta)^{-1}(c)} \{p.f \mid f \in F(\mathrm{Sig}(\Delta, p))\}$;

- for each part $c \in Cmp(\Delta)$, the attributes of $\mathrm{Sig}(\Delta, c)$ are all prefixed with c, i.e., if $a \in A(\mathrm{Sig}(\Delta, c))$, then $a = c.a_*$;

- for each connection $(p, p') \in Con(\Delta)$, the output events of port p are input events for port p', i.e., $O(\mathrm{Sig}(\Delta, p)) \subseteq I(\mathrm{Sig}(\Delta, p'))$.

We say that port $p \in Prt(\Delta)$ is *open* in Δ if there is no $p' \in Prt(\Delta)$ such that $(p, p') \in Con(\Delta)$; otherwise p is *connected*.

A *Composite Structure signature morphism* $\delta : \Delta \to \Delta'$ over EDHMLO consists of

- a function $Cmp(\delta) : Cmp(\Delta) \to Cmp(\Delta')$ mapping parts, together with
  - an EDHMLO-signature morphism $\mathrm{Sig}(\delta, c) : \mathrm{Sig}(\Delta, c) \to \mathrm{Sig}(\Delta', Cmp(\delta)(c))$ for each $c \in Cmp(\Sigma)$;

- a function $Prt(\delta) : Prt(\Delta) \to Prt(\Delta')$ mapping ports,
  - together with an EDHMLO-signature morphism $Prt(\delta)(p) : \mathrm{Sig}(\Delta, p) \to \mathrm{Sig}(\Delta', Prt(\delta)(p))$,

preserving

- the part c owning each port p, i.e., if $c = cmp(\Delta)(p)$, then $Cmp(\delta)(c) = cmp(\Delta')(Prt(\delta)(p))$;

- the connections, i.e., if $(p, p') \in Con(\Delta)$, then $(Prt(\delta)(p), Prt(\delta)(p')) \in Con(\Delta')$.

The category of cs(EDHMLO)-*signatures* $\mathrm{Sig}^{\mathrm{cs(EDHMLO)}}$ consists of the Composite Structure signatures and signature morphisms over EDHMLO.

Example 11.1 Considering the ATM example (Fig. 9.1), we would have parts

$$Cmp(\Delta) = \{\mathsf{atm}, \mathsf{bank}\}.$$

The signature for the atm component could be given by the following input events, output events and attributes:

$$
\begin{aligned}
|I(\mathrm{Sig}(\Delta,\mathrm{atm}))| = \{ \quad & \mathrm{userCom.card},\\
& \mathrm{userCom.PIN},\\
& \mathrm{bankCom.verified},\\
& \mathrm{bankCom.reenterPIN}\\
\}&\\
\bar{s}(I(\mathrm{Sig}(\Delta,\mathrm{atm}))) = \{ \quad & \mathrm{userCom.card} \mapsto (\mathbb{N}),\\
& \mathrm{userCom.PIN} \mapsto (),\\
& \mathrm{bankCom.verified} \mapsto (),\\
& \mathrm{bankCom.reenterPIN} \mapsto ()\\
\}&\\
|O(\mathrm{Sig}(\Delta,\mathrm{atm}))| = \{ \quad & \mathrm{bankCom.verify},\\
& \mathrm{userCom.ejectCard},\\
& \mathrm{userCom.keepCard}\\
\}&\\
\bar{s}(O(\mathrm{Sig}(\Delta,\mathrm{atm}))) = \{ \quad & \mathrm{bankCom.verify} \mapsto (\mathbb{N},\mathbb{N}),\\
& \mathrm{userCom.ejectCard} \mapsto (),\\
& \mathrm{userCom.keepCard} \mapsto ()\\
\}&\\
|A(\mathrm{Sig}(\Delta,\mathrm{atm}))| = \{ \quad & \mathrm{cardId},\\
& \mathrm{pin},\\
& \mathrm{trialsNum}\}\\
s(A(\mathrm{Sig}(\Delta,\mathrm{atm}))) = \{ \quad & \mathrm{cardId} \mapsto \mathbb{N},\\
& \mathrm{pin} \mapsto \mathbb{N},\\
& \mathrm{trialsNum} \mapsto \mathbb{N}\}
\end{aligned}
$$

The ports of the ATM Composite Structure could be given as:

$$Prt(\Delta) = \{\mathrm{atm.bankCom}, \mathrm{atm.userCom}, \mathrm{bank.atmCom}\}$$

The connectors could then be given by:

$$Con(\Delta) = \{(\mathrm{atm.bankCom}, \mathrm{bank.userCom})\}$$

To all this, we could add a completion event for each control state of each part and a completion port, connected to itself, for each part.

11.2 Structures

For an $\mathrm{cs}(\mathrm{EDHMLO})$-signature Δ, a Δ-*Composite Structure structure* (sic!) over EDHMLO is a family

$$\mathscr{C} \in (\mathscr{C}(c) \in |\mathrm{Mod}^{\mathrm{EDHMLO}}(\mathrm{Sig}(\Delta, c))|)_{c \in Cmp(\Delta)}$$

consisting of an EDHMLO-structure for each part c. The δ-*reduct* $\mathscr{C}'|\delta$ of a Δ'-Composite Structure structure $\mathscr{C}'$ over EDHMLO along a Composite Structure signature morphism $\delta : \Delta \to$

Δ' is computed componentwise as $(\mathscr{C}'(Cmp(\delta)(c))|\mathrm{Sig}(\delta,c))_{c\in Cmp(\Delta)}$. The Δ-Composite Structure structures form the discrete category $\mathrm{Mod}^{\mathrm{cs(EDHMLO)}}(\Delta)$ of $\mathrm{cs(EDHMLO)}$-*structures* over Δ. For each signature morphism $\delta:\Delta\to\Delta'$ in $\mathrm{Sig}^{\mathrm{cs(EDHMLO)}}$ the *δ-reduct functor*

$$\mathrm{Mod}^{\mathrm{cs(EDHMLO)}}(\delta):\mathrm{Mod}^{\mathrm{cs(EDHMLO)}}(\Delta')\to\mathrm{Mod}^{\mathrm{cs(EDHMLO)}}(\Delta)$$

is given by $\mathrm{Mod}^{\mathrm{cs(EDHMLO)}}(\delta)(\mathscr{C}')=\mathscr{C}'|\delta$.

In UML, State Machines organised in a Composite Structure communicate with each other by sending messages which are stored in event pools. A State Machine draws a message from its event pool, which is typically implemented as an event queue, and reacts to this message by firing one of its enabled transitions or by discarding it when no transition is enabled. This communication scheme is obtained for a Δ-Composite Structure structure $\mathscr{C}$ over EDHMLO by constructing an overall EDHMLO-structure over an EDHMLO-signature that reflects the parts, the ports, and the connections in its events and attributes, but includes explicit event queues as additional attributes. The overall EDHMLO-structure over this queue-based EDHMLO-signature then implements the selection of an event from a part's event queue, the reactions of this part to this event, and the distribution of the produced messages to the connected parts.

Formally, we construct a functor $\mathrm{Sig_q}:\mathrm{Sig}^{\mathrm{cs(EDHMLO)}}\to\mathrm{Sig}^{\mathrm{EDHMLO}}$ assigning to a Composite Structure signature Δ the *queue-based* event/data signature $\mathrm{Sig_q}(\Delta)=\bigcup_{c\in Cmp(\Delta)}(\mathrm{Sig}(\Delta,c)\cup\{\mathrm{q}_c:\hat{I}(\mathrm{Sig}(\Delta,c))^*\})$ and to a Composite Structure signature morphism the canonically corresponding event/data signature morphism. For a Composite Structure signature Δ and a part $c\in Cmp(\Delta)$ there is a natural signature embedding $\eta^{\mathrm{q}}_{\Delta,c}:\mathrm{Sig}(\Delta,c)\to\mathrm{Sig_q}(\Delta)$.

For a Δ-Composite Structure structure $\mathscr{C}$ we construct an overall $\mathrm{Sig_q}(\Delta)$-event/data structure $M_{\mathscr{C}}$ as follows: An overall configuration of $M_{\mathscr{C}}$ consists, for each part $c\in Cmp(\Delta)$, of an *event queue* $q(c)\in\hat{I}(\mathrm{Sig}(\Delta,c))^*$ stored in the attribute q_c and a part configuration $\gamma(c)\in\Gamma(\mathscr{C}(c))$; initially, all parts are in some of their initial configurations and all event queues are empty. The family of component control states forms the overal control state. In an overall configuration $(q(c),\gamma(c))_{c\in Cmp(\Delta)}$ an overall transition to another overall configuration $(q'(c),\gamma'(c))_{c\in Cmp(\Delta)}$ *reacts* to some $\hat{\imath}\in\hat{I}(\mathrm{Sig_q}(\Delta))$ and *outputs* some $\hat{O}\in\hat{O}(\mathrm{Sig_q}(\Delta))^*$. This $\hat{\imath}$ can either instantiate some input event $i\in I(\mathrm{Sig}(\Delta,p_*))$ of some of the open ports $p_*\in Prt(\Delta)$ with $c_*=cmp(\Delta)(p)$, or it is the head of the event queue of some $c_*\in Cmp(\Delta)$ such that $i\in I(\mathrm{Sig}(\Delta,c_*))$. In the latter case, $\hat{\imath}$ is removed from the event queue of c_*. In both cases, the reaction of part c_* is any transition $(\gamma(c_*),\gamma'_*)\in R(\mathscr{C}(c))_{\hat{\imath},\hat{O}}$ and overall $\gamma'=\gamma\{c_*\mapsto\gamma'_*\}$. Finally, all outputs $p.\hat{o}\in\hat{O}$ such that $(p,p')\in Con(\Delta)$ and $cmp(\Delta)(p')=c'$ are appended to the respective event queue of part c', while preserving their order. This defines a natural transformation $\mathrm{Mod}^{\mathrm{cs(EDHMLO)}}_{\mathrm{q}}:\mathrm{Mod}^{\mathrm{cs(EDHMLO)}}\dot{\to}\mathrm{Mod}^{\mathrm{EDHMLO}}\circ\mathrm{Sig_q}$ with $\mathrm{Mod}^{\mathrm{cs(EDHMLO)}}_{\mathrm{q},\Delta}(\mathscr{C})=M_{\mathscr{C}}$.

Example 11.2 Returning to the ATM example (Fig. 9.1), let us now construct its queue-based signature [1]. EDHMLO structures over this signature are then the Composite Structure structures for the ATM Composite Structure signature, here called Δ.

[1] We will omit some redundant information. Such omissions are indicated by dots (...).

First, we have have input and output events with their sorts:

$$\begin{aligned}
|I(\mathrm{Sig_q}(\Delta))| = \{ \quad & \mathrm{atm.userCom.card}, \\
& \mathrm{atm.userCom.PIN}, \\
& \mathrm{atm.bankCom.verified}, \\
& \mathrm{atm.bankCom.reenterPIN}, \\
& \mathrm{bank.atmCom.verify} \\
\} & \\
\overline{s}(I(\mathrm{Sig_q}(\Delta))) = \{ \quad & \dots \} \\
|O(\mathrm{Sig_q}(\Delta))| = \{ \quad & \mathrm{atm.bankCom.verify}(\mathrm{card}, \mathrm{pin}), \\
& \mathrm{atm.userCom.ejectCard}, \\
& \mathrm{atm.userCom.keepCard}, \\
& \mathrm{bank.atmCom.verified}, \\
& \mathrm{bank.atmCom.reenterPIN} \\
\} & \\
\overline{s}(O(\mathrm{Sig_q}(\Delta))) = \{ \quad & \dots \}
\end{aligned}$$

Then, we have the attributes, now crucially including the queues:

$$\begin{aligned}
|A(\mathrm{Sig_q}(\Delta))| = \{ \quad & \dots \} & & \\
s(A(\mathrm{Sig_q}(\Delta))) = \{ \quad & \mathrm{atm.cardId} & \mapsto \quad & \mathbb{N}, \\
& \mathrm{atm.pin} & \mapsto \quad & \mathbb{N}, \\
& \mathrm{atm.trialsNum} & \mapsto \quad & \mathbb{N}, \\
& \mathrm{bank.wasVerified} & \mapsto \quad & \mathbb{N}, \\
& q_{\mathrm{atm}} & \mapsto \quad & |I(\mathrm{Sig}(\Delta, \mathrm{atm}))|^*, \\
& q_{\mathrm{bank}} & \mapsto \quad & |I(\mathrm{Sig}(\Delta, \mathrm{bank}))|^*, \\
\} & & &
\end{aligned}$$

Like before, we could add completion events.

11.3 Sentences

$\mathrm{cs}(\mathrm{EDHMLO})$ inherits the event/data formulæ of EDHMLO:

$$\mathrm{Sen}^{\mathrm{cs(EDHMLO)}} = \mathrm{Sen}^{\mathrm{EDHMLO}} \circ \mathrm{Sig_q}$$

and the underlying $\mathcal{D}$, though extended by queue attributes. In particular, we have for a part $c \in Cmp(\Delta)$ that a transition sentence[2]

$$\langle\!| i : \phi \,/\!/\, O : \psi |\!\rangle \varrho$$

[2]As a reminder, this sentence means: In the current configuration there are valuations and a transition for the incoming message and the outgoing messages such that these valuations satisfy the transition formula ψ and that ϱ holds afterwards.

locally formulated for this part can be faithfully transferred to the global Composite Structure, abbreviating the embedding $\eta^{\mathrm{q}}_{\Delta,c}$ to η,

$$\begin{aligned}&\langle\!|\eta(i) : \mathscr{F}^{\mathcal{D}}_{A(\eta),X(i)}(\phi) \wedge (\mathrm{hd}(\mathrm{q}_c)\\ &\quad = I(\eta)(i) \vee \mathrm{open}_{\Delta,c}(I(\eta)(i)))|\!\rangle\\ &\qquad O(\eta)(O) : \mathscr{F}^{2\mathcal{D}}_{A(\eta),X(i)\cup X(O)}(\psi) \wedge \textstyle\bigwedge_{a\in A(\mathrm{Sig_q}(\Delta))\setminus(A(\mathrm{Sig}(\Delta,c))\cup\{\mathrm{q}_c|c\in Cmp(\Delta)\})} a = a' \wedge\\ &\qquad \mathrm{dist}_{\Delta,c}(I(\eta)(i), O(\eta)(O), (\mathrm{q}_c, \mathrm{q}'_c)_{c\in Cmp(\Delta)})\rangle\!|\,\mathrm{Sen}^{\mathrm{EDHMLO}}(\eta)(\varrho)\ ,\end{aligned}$$

where hd yields the head of a queue, open checks whether the part's port for the event is open, the frame condition $a = a'$ ranges over all attributes not pertaining to c or the queues, dist removes the input and distributes the outputs to the queues.

Binding of control states remains unchanged.

For control state assertions s in a component d, it is necessary to remember that an overall control state is a family (member of a cartesian product) of the component control states. Therefore, in a Composite Structure, s has to be expressed by the data sentence $s_d = c(\gamma(d))$, asserting that the component control state projected from s and that projected from the current component configuration are equal.

11.4 Satisfaction Relation and Satisfaction Condition

The *satisfaction relation* is an extension of that of EDHMLO: $\mathscr{C} \models^{\mathrm{cs(EDHMLO)}}_{\Delta} \varrho$ if, and only if, $\mathrm{Mod}^{\mathrm{cs(EDHMLO)}}_{\mathrm{q},\Delta}(\mathscr{C}) \models^{\mathrm{EDHMLO}}_{\mathrm{Sig_q}(\Delta)} \varrho$.

Theorem 11.3 $(\mathrm{Sig}^{\mathrm{cs(EDHMLO)}}, \mathrm{Mod}^{\mathrm{cs(EDHMLO)}}, \mathrm{Sen}^{\mathrm{cs(EDHMLO)}}, \models^{\mathrm{cs(EDHMLO)}})$ is an institution.

Proof. This directly carries over from EDHMLO, as cs(EDHMLO)inherits everything except signatures directly from EDHMLO. However, we will unfold the definitions explicitly:

$$\begin{aligned}&\mathscr{C}'|\sigma \models^{\mathrm{cs(EDHMLO)}}_{\Delta} \varrho\\ \Leftrightarrow\quad &\{\ \text{def. } \models^{\mathrm{cs(EDHMLO)}}\ \}\\ &\mathrm{Mod}^{\mathrm{cs(EDHMLO)}}_{\mathrm{q},\Delta}(\mathscr{C}'|\sigma) \models^{\mathrm{EDHMLO}}_{\Sigma} \varrho\\ \Leftrightarrow\quad &\{\ \text{def. } \mathrm{Mod}^{\mathrm{cs(EDHMLO)}}_{\mathrm{q}}\ \}\\ &\mathrm{Mod}^{\mathrm{cs(EDHMLO)}}_{\mathrm{q},\Delta}(\mathscr{C}')|\sigma \models^{\mathrm{EDHMLO}}_{\Sigma} \varrho\\ \Leftrightarrow\quad &\{\ \text{satisfaction condition of EDHMLO}\ \}\\ &\mathrm{Mod}^{\mathrm{cs(EDHMLO)}}_{\mathrm{q},\Delta}(\mathscr{C}') \models^{\mathrm{EDHMLO}}_{\Sigma} \varrho\\ \Leftrightarrow\quad &\{\ \text{def. } \mathrm{Sig_q}\ \}\\ &\mathscr{C}' \models^{\mathrm{cs(EDHMLO)}}_{\Delta'} \mathrm{Sig_q}(\sigma)(\varrho)\\ \Leftrightarrow\quad &\{\ \text{def. sentence translation}\ \}\\ &\mathscr{C}' \models^{\mathrm{cs(EDHMLO)}}_{\Delta'} \sigma(\varrho)\end{aligned}$$

□

Corollary 11.4 The comorphism to CASL also directly carries over from EDHMLO to $\mathrm{CS}(\mathrm{EDHMLO})$ by prefixing each of its components with Sig_q.

Chapter 12

Implementing our Institutions and Comorphisms in HeTS

Contents

In this chapter, we describe the implementations of our institutions and comorphisms in the Heterogeneous Toolset HeTS. To be precise, we implemented the languages UMLState (State Machines without outputs), UMLStateO (State Machines with outputs) and UMLComp (Composite Structures), and, for each of the above, a translation to Casl.

The concepts underlying these implementations are the result of joint work with our coauthors in [RBKR20, RKR22]. The implementation itself and its description here are exclusively the work of the present author.

The precise UMLState input language we implement is specified in App. A. We have crafted it to resemble PlantUML. For UMLStateO, the changes to the input language are discussed in this chapter as part of a discussion of the design choices for our implementation. A heterogeneous specification involving UMLState is given in App. C, another one for an example system involving UMLStateO and UMLComp is given in App. D. In the case of UMLState, the specification is the one for our Counter example on which we actually perform our proofs in Chapter 13. In the case of UMLStateO and UMLComp, we show the ATM example. For this, we have performed proofs on a manual translation into KIV, for reasons we explain in Chapter 13. Figure 12.1 shows the development graphs resulting from parsing and analysing these two specifications. Part of the output of the automatic translation for the ATM example can be seen in Fig. 12.2.

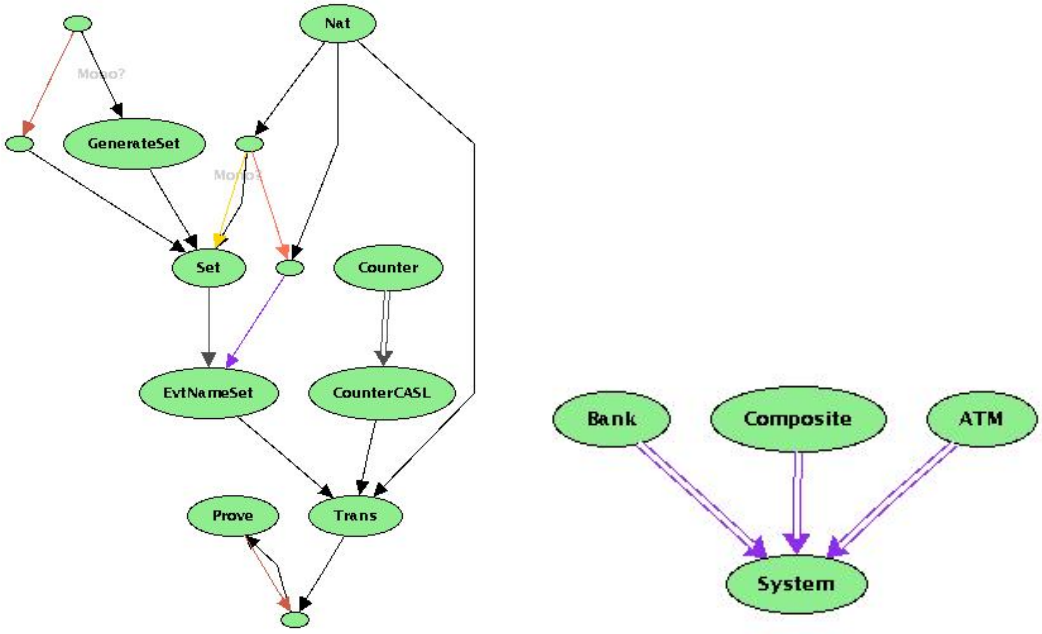


Figure 12.1: The development graphs shown by HETS after parsing the example files from Apps. C and D.

```
pred transOfATM : ConfOfATM * Msg * MsgList * ConfOfATM
pred transOfATMSystem : ConfOfATMSystem * Msg * MsgList *
                        ConfOfATMSystem
pred transOfBank : ConfOfBank * Msg * MsgList * ConfOfBank

forall g : ConfOfBank
. init(g) <=> ctrl(g) = stOfIdleOfBank /\ true            %(init)%

%% free
generated types
Msg ::= msg(Port; Evt);
ConfOfBank ::= conOfConfOfBank(CtrlOfBank; Nat)     %(free_types)%

forall g' : ConfOfBank
. ctrl(g') = stOfIdleOfBank
  /\ reachable2(conOfMsgNameOfverify
                + (conOfMsgNameOfcomplEscFIdle
                   + (conOfMsgNameOfcomplEscFVeriFail
                      + (conOfMsgNameOfcomplEscFVeriSuccess
                         + (conOfMsgNameOfcomplEscFVerifying + {})))),
                conOfMsgNameOfreenterPIN
                + (conOfMsgNameOfverified
                   + (conOfMsgNameOfcomplEscFIdle
                      + (conOfMsgNameOfcomplEscFVeriFail
                         + (conOfMsgNameOfcomplEscFVeriSuccess
                            + (conOfMsgNameOfcomplEscFVerifying + {}))))),
                g')
  => true
     => exists g'' : ConfOfBank
        . (trans(g', msg(conOfPortOfatmCom, conOfEvtOfverify), [], g'')
           /\ attrOfwasVerifiedOfBank(g'') = 0)
          /\ stOfVerifyingOfBank = ctrl(g'')           %(machine)%
```

Figure 12.2: Part of the output shown by HETS for the resulting CASL specification `System` from App. D.

To give the reader an idea of both the size of the system into which we integrated our work and the code itself which we wrote in that context, we report some line counts: The Haskell source files of HETS after our changes have 284976 lines of code, including everything (comments, empty lines, etc.). Of these 1005 lines are in the files specifically dedicated to UMLSTATE and its translation to CASL, 1285 lines are in the files dedicated to UMLSTATEO and its translation (adapted from the ones for UMLSTATE) and 738 lines for UMLCOMP and its translation. We made minor changes to other parts of the HETS codebase to enable HETS to use our code. Moreover, we use the declarative nix package management language to reproducibly install dependencies. Our nix files contain a total of 280 lines.

12.1 Implementation of the UMLSTATE Syntax in HETS

A syntax in HETS is implemented as an instance of the `Syntax` typeclass by giving a parser for basic specifications and a pretty printer, the latter being undefined in our case.

```
instance Syntax UMLState BASIC_SPEC Token () () where
  parsersAndPrinters UMLState = makeDefault (basicSpec [], pretty)
```

We define the abstract syntax in a style similar to a dedicated language for grammars. To be precise, for each non-terminal symbol we define an algebraic datatype for its syntax trees, e.g.:

```
data BASIC_SPEC = Basic [BASIC_ITEMS] deriving (Eq,Ord,Show,D.Data)
data BASIC_ITEMS = SigB SIG_ITEMS
                 | VarB VAR_ITEMS
                 | SenB SEN_ITEMS
                 | EvtB EVENT_ITEMS
                 deriving (Eq,Ord,Show,D.Data)
```

That is, a `BASIC_SPEC` is given by a list of `BASIC_ITEMS`, which can be either `SIG_ITEMS`, `VAR_ITEMS`, `SEN_ITEMS` or `EVENT_ITEMS`. The `deriving` clauses add automatically derivable instances for some Haskell typeclasses. Here, they give us support for deciding equality and order among elements, for showing elements as strings and for a certain kind of generic programming.

Then we define a `Parsec` parser for each non-terminal, e.g.:

```
basicItems :: Parsec [Char] st BASIC_ITEMS
basicItems = do SigB <$> sigItems
        <|> do SenB . TransB <$> transItem
        <|> do SenB <$> (key "init" >> (InitB  <$> (stateP  <<
   asSeparator ":") <*> guardP))
        <|> do EvtB <$> evtItems
        <|> do VarB <$> varItems
```

The above is the parallel composition of the parsers for the different constructor cases of the `BASIC_ITEMS` type.

```
1  logic UMLState
2  spec Counter =
3    var cnt;
4
5    event inc(k);
6    event reset;
7
8    states s1,s2;
9
10   init s1 : [cnt == 0];
11   trans s1 --> s1 : inc(k) [cnt+k<4]/{cnt:=cnt+k};
12   trans s1 --> s2 : inc(k) [cnt+k==4]/{cnt:=cnt+k};
13   trans s2 --> s1 : reset [cnt==4]/{cnt:=0}
14 end
```

```
Analyzing file file:///home/tobias/src/csmarkus/Tobias/PhD_new/thesis//specs/CounterScoping.het as
 library CounterScoping
logic UMLState
Analyzing spec Counter
file:///home/tobias/src/csmarkus/Tobias/PhD_new/thesis//specs/CounterScoping.het:4.3-14.1:
*** Error:
assignment to undeclared variable
hets: user error (Stopped due to errors)
```

Figure 12.3: A brief excerpt from our Counter specification, together with a screenshot of the scoping error which results from removing line 3. The specification is handwritten in our language UMLSTATE, which we crafted to resemble PlantUML. It has not been generated from a graphical notation, rather, the philosophy here would be to generate the graphical notation from the textual one. That said, it could be possible in the future to implement a (necessarily partial) translation from the XML-based XMI interchange format for UML.

In general, we algebraically build up the parsers from combinators (cf. Sect. 4.4). This approach to parsing allows us again to closely follow the structure of the grammar, while of course adding some operational details.

12.2 Implementing Static Analysis for UMLSTATE

After parsing, we perform static analysis. Again, we give our implementation by instantiating a typeclass provided by HETS:

```
instance StaticAnalysis UMLState BASIC_SPEC EDHML () () Library
    Morphism Token () where
  basic_analysis _ = Just $ \ (spec,sign,annos) -> do
    (spec',lib) <- runStateT (ana_BASIC_SPEC spec) mempty
    let sign   = lib
        symSet = sign2SymSet $ lib2Sign lib
        extSign = ExtSign sign symSet
        annos  = []
    return (spec', extSign, annos)
```

The above is an implementation of the `basic_analysis` method of the `StaticAnalysis` typeclass of HETS. At the core of this is the stateful action `ana_BASIC_SPEC spec`, where `spec` is the parse tree of a basic specification. The Haskell library function `runStateT` takes such a stateful action together with an initial state (here: `mempty`, i.e., the empty signature) and returns a pair of the action's return value and final state. The remainder of the above code is concerned with computing a symbol set for our signature and an (empty) list of annotations, as the method's return type requires.

The goal of our static analysis is to report scoping errors (see, e.g., Fig. 12.3) and to extract signature items from the basic specification. The signature, which is the state and result of the static analysis, is of the following type[1].

```
data Library = Library
  { statesL :: Set STATE
  , attrL   :: Set VAR_NAME
  , actsL   :: Map (EVENT_NAME, Arity) EVENT_ITEM
  , initL   :: [(STATE, GUARD)]
  , transL  :: [TRANS_ITEM]
  } deriving (Eq,Ord,Show)
```

These UMLSTATE signatures correspond to the simple UML State Machines of Sect. 7.2.2 together with the underlying event/data signature. The following points are noteworthy:

– The event/data signature is realised as a map/dictionary data structure indexed by `EVENT_NAME`s and `Arities`, the number of arguments. The names of the argument variables are not considered part of their identity – where two `EVENT_NAME`s match except for the argument names, one of them will be dropped.

– Instead of one initial control state and initial state predicate, the formula contains a list of pairs of such. The reason is that CASL requires us to be able to combine two signatures by a monoidal operation. Changing to a list here seemed the best way to fulfil that requirement.

Like in the case of parsing, we can largely just follow the structure of the grammar, with state and errors being passed along implicitly using a monad:

```
ana_BASIC_ITEMS :: BASIC_ITEMS -> Check BASIC_ITEMS
ana_BASIC_ITEMS (SigB items) = SigB <$> ana_SIG_ITEMS items
ana_BASIC_ITEMS (SenB items) = SenB <$> ana_SEN_ITEMS items
ana_BASIC_ITEMS (EvtB items) = EvtB <$> ana_EVENT_ITEMS items
ana_BASIC_ITEMS (VarB items) = VarB <$> ana_VAR_ITEMS items
```

[1] In UMLSTATE, we named this type `Library`, which is unfortunate, as it does not correspond to the library concept of CASL. In UMLSTATEO, the corresponding type is known as `Sign`. Moreover, in UMLSTATE we have a `Sign` type which only contains those entries of `Library` that define symbols, i.e., not the initialisation and transition "sentences". Note however, that these are not proper sentences, as their effect is not limited to the exclusion of models. They do belong in the signature, and in UMLSTATEO we abolish the distinction.

Mainly in the leaves of the syntax tree do we need to explicitly access the state, for both correctness checks and to compute the resulting library. E.g., the following code analyses an attribute declaration by first retrieving the state (the library computed so far), raising an error if an attribute with the same name has already been declared and finally, if there is no error, saving a new state with the variable added.

```
ana_VAR_ITEM :: VAR_NAME -> Check VAR_NAME
ana_VAR_ITEM var = do
  lib <- get
  let vars = attrL lib
  when (var `Set.member` vars) $ errC ("variable declared twice: " ++
    show var)
  put $ lib {
    attrL = var `Set.insert` vars
  }
  return var
```

12.3 Translation of State Machines

Our translation algorithm into EDHML can be written in Haskell on a similar level of abstraction as it would be in pseudocode, due to the availability of list comprehensions and defined operators, including infix operators. As an example, we show the definition of EDHML sentences as an inductive data type and the expression for constructing the fin sentences. The latter corresponds to the fin family in line 13 and 14 of Alg. 1). Note, however, that the fin in the implementation takes the conjunction of the fins over all control states c, and explicitly takes arguments that were taken from the surrounding context in Alg. 1. Moreover, in the below we make use of a function complements that generates a list of the complemented pairs of sets used in Alg. 1. The plural variable names here (bs,es) correspond to the upper case variables there.

```
data EDHML = DtSen FORMULA
           | St STATE
           | Binding STATE EDHML
           | At EVENT_NAMES STATE EDHML
           | Box EVENT_NAMES EDHML
           | DiaEE EVENT_ITEM FORMULA EDHML
                -- exists valuation and transition ...
           | DiaAE EVENT_ITEM FORMULA FORMULA EDHML
                -- for each valuation satisfying phi
                --  there exists a transition ...
           | Not EDHML
           | And EDHML EDHML
           | TrueE
           deriving (Eq,Ord,Show,D.Data)

```

```
16
17 fin :: EVENT_NAMES
18     -> [STATE]
19     -> ( (STATE, EVENT_NAME, Int) -> [(FORMULA, FORMULA, STATE)]
20        )
21     -> [EVENT_ITEM]
22     -> EDHML
23 fin allEvts bs im2 es = conjunctE
24   [ At allEvts c
25       $ conjunctE [ boxAA e (
26                       (
27                         conjunctF [ phi :/\ psi
28                                   | (phi, psi, c') <- ps
29                                   ]
30                       ) :/\ NotF (
31                         disjunctF  [ phi :/\ psi
32                                    | (phi, psi, c') <- nps
33                                    ]
34                       )
35                     ) $ disjunctE [ St c'
36                                   | (phi, psi, c') <- ps
37                                   ]
38                   | e <- es, (ps, nps) <- complements c e im2
39                   ]
40   | c <- bs
41   ]
```

The translation we register with HETS goes directly from UMLSTATE to CASL. We generate CASL signatures from the result of our static analysis. We have implemented a translation from EDHML to CASL sentences, which we apply to the EDHML sentence generated by our usual algorithm. As explained in Sect. 13.1, we have implemented some modifications to the formal comorphism that make proofs easier, such as Skolemisation and eliminating double negations. Moreover, as we have to generate basic rather than structured specifications, we leave imports from the standard library up to the user. For an example, see the specification in the appendices, i.e., App. C for the Counter machine and App. D for the ATM system. This also allows limiting the axioms that are brought into scope, which is often necessary for feasible fully automatic proving.

We then use these parts to define an instance of the `Comorphism` typeclass, just as we did before with the `Syntax` typeclass.

12.4 Design of Composite Structures and State Machines with Output

Our implementation of UMLSTATEO is similar to that of UMLSTATE, with the main difference that we have to thread outputs through everything. To allow easier proofs and interoperability with UMLCOMP and other formalisms, we have further made the following design choices:

We now introduce a notion of port names and of messages. However, we wish to keep treating messages just like plain events as far as possible. In the input language, a message, consisting of a port name and an event name, is always treated as one unit. In the translation to CASL, we need separate types for events and messages, however, to enable the combination of State Machines with Composite Structures: Composite Structures consider events to be an opaque type, defined by State Machines, but they need to be able to change a message's sending port to its receiving port before inserting the message into the receiver's queue.

We further introduce a name for each machine specification (considered the machine *type* name), which we have to pass into most computations, because we include it in the names of the transition predicate, as well as most predicates and sorts. However, we have decided to *not* include the name in the event and message sorts and their operations, with the effect that those signature elements are shared between all State Machines and Composite Structures unless they are explicitly renamed. This removes the need of applying and reasoning about type conversions when a message is produced by one machine and consumed by another. We likewise give a name to a composite structure and to each of its components (machine instances).

Given these names, we generate a configuration sort, an initialisation predicate and a transition predicate, respectively, for each State Machine and the Composite Structure. These are identified using CASL's "same name, same thing" principle: They are named by appending the machine or structure name to *ConfOf*, *initOf*, and *transOf*. Moreover, we define a predicate with name *distOf* prepended to the Composite Structure name, which relates a list of output events to message queues before and after distributing those events.

As in the semantics, the configurations of the Composite Structure are defined to consist of a configuration and a message queue for each machine. By handling message queues in the Composite Structure, we simplify the machine definitions and ease separate reasoning about the communication and the internal behaviour of components.

The Composite Structure is in its initial configuration iff all the machines are in their respective initial configurations and, in addition, the queues are empty.

The Composite Structure makes a transition *iff*:

- the machine to which the input message is addressed can make a transition for this trigger for the indicated configurations and outputs,
- the configurations of the other machines remain unchanged,
- events are added to the queues as the *distOf* predicate and the output list require, and
- the trigger message belongs to an external port or is removed from the appropriate queue.

Note that this behaviour of Composite Structures directly corresponds to that specified in Chapter 11, with the State Machines factored out as described here.

Messages from or to the outside do not interact with queues at all. They appear in the transition relations, but are otherwise ignored by the Composite Structure. As the Composite Structure itself is given by the same interface as a State Machine, it can itself become part of other Composite Structures which would then handle its external communication.

12.4.1 Extending the UMLSTATE grammar to UMLSTATEO

We introduce MESSAGE_NAMEs, each consisting of a PORT_NAME and an EVENT_NAME. These new MESSAGE_NAMEs are used wherever we were using EVENT_NAMEs so far.

$$\begin{aligned} \texttt{MESSAGE\_NAME} &::= \texttt{PORT\_NAME}\ \texttt{EVENT\_NAME} \\ \texttt{PORT\_NAME} &::= \texttt{Token} \end{aligned}$$

To the BASIC_SPEC we add a MACHINE_NAME to identify a machine in the context of a Composite Structure, so

$$\texttt{BASIC\_SPEC} ::= \texttt{basic-spec}\,\langle \texttt{BASIC\_ITEMS}^{+}\,\rangle$$

becomes:

$$\begin{aligned} \texttt{MACHINE\_NAME} &::= \texttt{Token} \\ \texttt{BASIC\_SPEC} &::= \texttt{MACHINE\_NAME}\,\langle \texttt{BASIC\_ITEMS}^{+}\rangle \end{aligned}$$

In BASIC_ITEMS, we now have MESSAGE_ITEMS instead of EVENT_ITEMS.

$$\begin{aligned} \texttt{BASIC\_ITEMS} ::= {} & \texttt{sig-its}\,\langle \texttt{SIG\_ITEMS}\rangle \\ \mid {} & \texttt{var-its}\,\langle \texttt{VAR\_ITEMS}\rangle \\ \mid {} & \texttt{evt-its}\,\langle \texttt{MESSAGE\_ITEMS}\rangle \\ \mid {} & \texttt{sen-its}\,\langle \texttt{SEN\_ITEMS}\rangle \end{aligned}$$

EVENT_ITEMS become MESSAGE_ITEMS and gain a direction, so that in addition to inputs we can now also handle outputs. This turns:

$$\begin{aligned} \texttt{EVENT\_ITEMS} &::= \texttt{evt-it}\,\langle \texttt{EVENT\_ITEM}^{+}\,\rangle \\ \texttt{EVENT\_ITEM} &::= \texttt{EVENT\_NAME}\ \texttt{VAR\_NAME}^{+}\ \texttt{VAR\_NAME} \end{aligned}$$

into:

$$\begin{aligned} \texttt{MESSAGE\_ITEMS} &::= (\texttt{input} \mid \texttt{output})\,\langle \texttt{MESSAGE\_ITEM}^{+}\rangle \\ \texttt{MESSAGE\_ITEM} &::= \texttt{MESSAGE\_NAME}\ \texttt{VAR\_NAME}^{+}\ \texttt{VAR\_NAME} \end{aligned}$$

Triggers become optional as we are now handling completion events, although without giving them a special priority. So we turn:

$$\begin{aligned} \texttt{TRANS\_LABEL} &::= \texttt{TRIGGER}\,\langle \texttt{GUARD}\rangle\,\langle \texttt{ACTIONS}\rangle \\ \texttt{TRIGGER} &::= \texttt{EVENT\_ITEM} \end{aligned}$$

into:

$$\begin{aligned} \texttt{TRANS\_LABEL} &::= \langle \texttt{TRIGGER} \rangle \langle \texttt{GUARD} \rangle \langle \texttt{ACTIONS} \rangle \\ \texttt{TRIGGER} &::= \texttt{MESSAGE\_ITEM} \end{aligned}$$

We add another kind of ACTION beside assignment, namely that of sending a MESSAGE, turning:

$$\texttt{ACTION} ::= \texttt{VAR\_NAME TERM}$$

into:

$$\begin{aligned} \texttt{ACTION} &::= \texttt{assign-act VAR\_NAME TERM} \\ &\mid \texttt{send-act MESSAGE\_INSTANCE} \\ \texttt{MESSAGE\_INSTANCE} &::= \texttt{MESSAGE\_NAME} \langle \texttt{TERM}^{+} \rangle \end{aligned}$$

The resulting signatures for UMLSTATEO are now of the form:

```
data Sign = Sign
  { nameS   :: Set MACHINE_NAME
  , statesS :: Set STATE
  , attrS   :: Set VAR_NAME
  , trigS   :: Map (MESSAGE_NAME, Arity) MESSAGE_ITEM
  , actsS   :: Map (MESSAGE_NAME, Arity) MESSAGE_ITEM
  , initS   :: [(STATE, GUARD)]
  , transS  :: [TRANS_ITEM]
  } deriving (Eq,Ord,Show)
```

Composite Structure signatures take the form:

```
data Sign = Sign
  { nameS         :: [COMP_NAME]
  , machinesS     :: Map MACHINE_NAME MACHINE_TYPE
  , machineTysS   :: Set MACHINE_TYPE
  , exportsS      :: Map PORT_NAME MACHINE_NAME
  , connsS        :: Map PORT_NAME (MACHINE_NAME, MACHINE_NAME,
    PORT_NAME)
  } deriving (Eq,Show,Ord)
```

The reader may wish to take a moment to compare the two before we continue in the next section with an explanation how to interface with our languages.

12.5 Interfacing with UMLStateO and UMLComp

The design we have described makes it possible to interface between UMLStateO, UMLComp and other formalisms. The recommended way to do so is by translating to CASL and combining the resulting specifications using CASL's structuring constructs. The best way to connect to UMLStateO is through UMLComp, or by the same interface used by UMLComp.

UMLComp can connect with any formalisation that axiomatises compatible initialisation and transition predicates. In the following, we will show the sort, function and predicate symbols arising from the translation of a Composite Structure using our comorphism. For compatibility and maintainability, they can and should be imported from `UMLComp.Logic_UMLComp`. By excerpts from that module, we will now acquaint the reader with the interface our Composite Structures expose through the comorphism.

There are the following sort symbols, where each exists either once per machine type (not machine instance!) and once for the entire Composite Structure, or just once for the Composite Structure[2]. The below list contains each sort symbol paired with an (empty) set of supersorts. The list specifies a binary relation of sorts, the reflexive transitive closure of which will be CASL's subsorting relation for our specification.

```
            -- per machine type/composTy
            [ (s tyName, Set.empty)
            | tyName <- composTy : machTys
            , s <- [ confSort ]
            ] ++
            -- just once (per composTy)
            [ (s, Set.empty)
            | s <- [ portSort
                   , evtSort
                   , msgSort
                   , msgListSort
                   , msgQueueSort
                   ]
            ]
```

The following predicate symbols are generated, where, as with the sorts, each symbol exists either for the Composite Structure and each machine type or just once for the entire Composite Structure. The list pairs each predicate symbol with the (singleton) list of types to which it can belong.

```
              -- per machine type/composTy
              [ (pname, Set.fromList [pty])
              | tyName <- composTy : machTys
              , (pname, pty) <- [ initP  tyName
                                , transP tyName
                                ]
```

[2]Without renaming, they are even global, i.e., shared between Composite Structures.

```
            ] ++
            -- just once (per composTy)
            [ (pname, Set.fromList [pty])
            | (pname, pty) <- [ distP composTy $ length machs ]
            ]
```

Moreover, the comorphism generates operation symbols. Like the predicate symbols, these appear paired with the set of their possible types. The operations symbols are now for individual machines (not types) or for the entire structure. Note the reference to constructors for free types, a list of which will follow.

```
            -- per individual machine (and composTy)
            [ (oname, Set.fromList [oty])
            | (machName,_) <- machs
            , (oname,oty)  <- [ queueProjOp composTy machName
                              ]
            ] ++
            -- just once (per composTy)
            [ (oname, Set.fromList [oty])
            | (oname,oty) <- [ con
                             | (_,cons) <- constructors
                             , con <- cons
                             ]
                           ++ [ dequeueOp ]
                           ++ ( portConOp <$> portNames )
            ]
```

The following are the types and constructors for which the translation of a Composite Structure will contain the symbols and freeness axioms:

```
    -- Constructors for free types.
    -- Note that machine specific types,
    --   as well as event types,
    --   are not free as far as the composite structure is concerned.
    constructors = [ ( (confSort composTy)
                     , [confConOp composTy machs]
                     )
                   , ( msgSort
                     , [msgConOp]
                     )
                   , ( msgListSort
                     , [nilOp, consOp]
                     )
                   , ( msgQueueSort
                     , [emptyQueueOp, enqueueOp]
                     )
```

```
17                                    ]
```

For an example specification that makes use of this interface for the case of UMLStateO and UMLComp, we refer the reader once more to App. D.

12.6 How to Obtain our Implementation

Our implementation is available as a fork[3] of the HeTS git repository[4]. The current commit at the time of writing has the id dca77099c442c0df67111c54e5aa78f19d685609. We intend to contribute our work to upstream HeTS, but also to keep the version on which this thesis is based available in our fork. Within our fork, the file *README.md* contains information on how to build HeTS while recursively using the same versions of the dependencies that we also built with.

[3] https://github.com/rosento/Hets/
[4] https://github.com/spechub/Hets/

Chapter 13

Verification Examples

Contents

In this chapter, we discuss two verification examples which we carried out using our methodology, formalisms and, to an extent, our newly developed tools. The material on the counter machine has been published in similar form as part of [RBKR20]. The material on the ATM scenario has been published as part of [RKR22].

The examples and verification goals are the result of discussions with Alexander Knapp and Markus Roggenbach. The development of our general formalisms was in part guided by formalisations of the examples. I formalised the examples and carried out the proofs, including the development of necessary auxiliary lemmas, our trick for inductive verification without tool-support for the structured free construct, and the manual translation to KIV.

The ATM example is a variation on a popular theme; see, e.g., [KMR15].

13.1 The Counter Machine: Proofs about State Machines without Outputs with HETS and SPASS

We implemented the translation of simple UML State Machines into CASL specifications within the heterogeneous toolset HETS [MML07b]. Based on this translation we explain how to prove properties symbolically in the automated theorem prover SPASS [WDF+09] for our running example of a counter.

Figure 13.1 shows the CASL specification representing the State Machine from Fig. 7.1, extended by a proof obligation %(Safe)% and proof infrastructure for it. We want to prove the safety property

> "cnt never exceeds 4"

spec COUNTER = TRANS
then pred $invar(g : Conf) \Leftrightarrow (c(g) = s1 \wedge cnt(g) \leq 4) \vee (c(g) = s2 \wedge cnt(g) \leq 4)$
%% induction scheme for "reachable" predicate, instantiated for "invar":
$\cdot ((\forall\ g : Conf \cdot init(g) \Rightarrow invar(g))$
$\wedge\ \forall\ g, g' : Conf; e : Evt$
$\cdot (reachable(g) \Rightarrow invar(g)) \wedge reachable(g) \wedge trans(g, e, g') \Rightarrow invar(g'))$
$\Rightarrow \forall\ g : Conf \cdot reachable(g) \Rightarrow invar(g)$
then ... *machine axioms* ...
then %implies
%% the safety assertion for our counter:
$\forall\ g : Conf \cdot reachable(g) \Rightarrow cnt(g) \leq 4$ %(Safe)%
%% steering SPASS with case distinction lemmas, could be generated algorithmically:
$\forall\ g, g' : Conf; e : Evt; k : Nat$
$\cdot\ init(g) \Rightarrow invar(g)$ %(InvarInit)%
$\cdot\ (reachable(g) \Rightarrow invar(g)) \wedge reachable(g) \wedge trans(g, e, g') \wedge e = reset$
$\Rightarrow invar(g')$ %(InvarReset)%
$\cdot\ (reachable(g) \Rightarrow invar(g)) \wedge reachable(g) \wedge trans(g, e, g') \wedge e = inc(k)$
$\Rightarrow invar(g')$ %(InvarInc)%
$\cdot\ (reachable(g) \Rightarrow invar(g)) \wedge reachable(g) \wedge trans(g, e, g')$
$\Rightarrow invar(g')$ %(InvarStep)%
$\cdot\ invar(g) \Rightarrow cnt(g) \leq 4$ %(InvarImpliesSafe)%

Figure 13.1: A partial CASL specification of the counter. Appendix C shows a full, heterogeneous version.

using the automated theorem prover SPASS.

The CASL specification COUNTER imports a specification TRANS which instantiates the generic frame translating EDHML into CASL, cf. Fig. 8.1. However, the first-order theorem prover SPASS does not support CASL's structured free construct that we use for expressing reachability. For invariance properties this deficiency can be circumvented by loosely specifying *reachable* (i.e., omitting the keyword `free`), introducing a predicate *invar*, and adding a first-order induction axiom. This means that we have to establish the safety property for a larger model class than we would have with freeness. When carrying out symbolic reasoning for invariants referring to a single configuration, the presented induction axiom suffices. Other properties would require more involved induction axioms, e.g., referring to several configurations.

Then the specification provides the machine axioms as stated (partially) in Thm. 7.13. The axioms following the *%implies* directive are treated as proof obligations. We first state the safety property that we wish to establish: in all reachable configurations, the counter value is less or equal 4 – %(Safe)%. The remainder steers the proving process in SPASS by providing suitable case distinctions. For invariants referring to a single configuration, these could also be generated automatically based on the transition structure of the state machine, although we have written them by hand.

As a proof of concept, we automatically verified the safety property from Fig. 13.1 in SPASS. In this experiment, we performed some optimising, semantics-preserving logical transformations

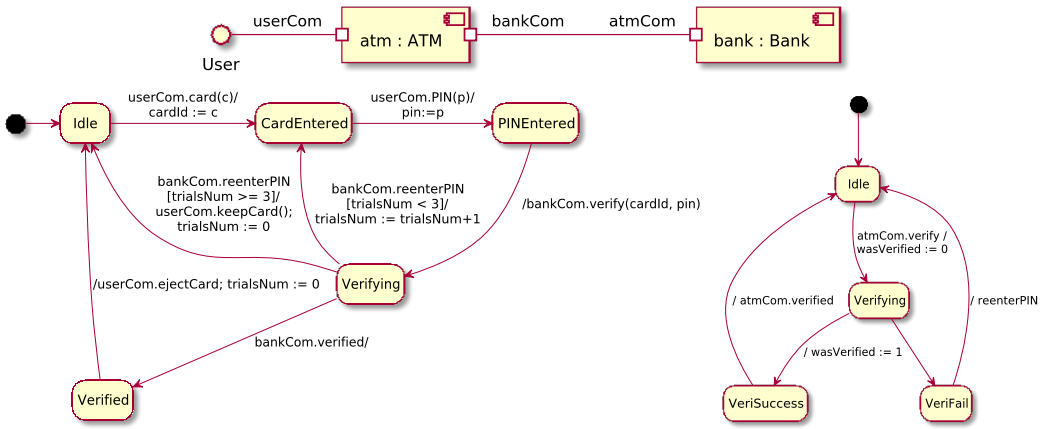


Figure 13.2: UML diagrams for the `ATM` example verified in Sect. 13.2 (implicit completion events omitted): Composite Structure diagram: top; State Machines: left `ATM`, right `Bank`.

on the result of applying the comorphism, to make the specification more digestible to the theorem prover. These transformations include the removal of double negations, splitting a conjunction into separate axioms, and turning existentially quantified control states into constants by Skolemisation. Our automatic translation as implemented includes these transformations.

13.2 The ATM Scenario: Verifying Composite Structures with KIV

Recall the ATM example. For convenience, we repeat it's diagrams in Fig. 13.2 and briefly recall the scenario: An `ATM` and a `Bank` communicate to enable a user to withdraw money. This withdrawal should only be possible after the bank has confirmed that the user has entered the correct bank card and PIN.

We formalised[1,2] the State Machines for the `Bank` and the `ATM` as well as their communication in CASL. We then set out to show the safety property:

$$C(\gamma(\texttt{atm})) = \text{Verified} \rightarrow \text{wasVerified}(\gamma(\texttt{bank})) = 1$$

This states that, whenever `atm` reaches the control state Verified, the `bank`'s attribute wasVerified has the value 1. To recall some notation: $\gamma(c)$ projects a component c's configuration from the system configuration; C projects the control state from a part's configuration; an attribute (here: wasVerified) gives us a projection from it's parts configuration.

We show the safety property by inductive verification, as justified by Prop. 10.3, using a stronger invariant. We first tried to show the preservation of said invariant using fully automatic provers connected to HETS. Unfortunately, the handling of freely generated datatypes in this tool

[1]The full formalisation of the ATM example, including an importable KIV project, is available in the digital appendix to [RKR22] at `https://rosento.github.io/2021-paper-composite/`.

[2]A heterogeneous formalisation of the ATM example using our languages UMLSTATE and UMLCOMP can be found in App. D. However, this was not directly used for automatic verification for the reasons outlined in this section.

combination turned out not to be efficient enough without a laborious manual axiom selection that would defeat our goal of automation, leading us to employ the prover KIV.

KIV [EPS+15] is an interactive theorem prover for program verification, which we used effectively as an automatic prover for first-order logic with freely generated types. There is currently no integration of KIV into HETS, nor is there, to our knowledge, a formulation of its logic as an institution. However, for the fragment of KIV we are using we are confident that such a formalisation would be tedious, but straightforward. Consequently, we resorted to a manual translation to KIV, which we judged sufficient for demonstration purposes.

With our process clarified, we can define a KIV predicate for our safety property:

```
safe-def: safe(g) ↔ (ctrl(caConf(g)) = Verified → wasVerified(cbConf(g)) = 1); used for: s, ls;
```

The above introduces an axiom `safe-def` defining the predicate `safe` and marks the axiom for use as a simplifier rule (`s`) and a local simplifier rule (`ls`) for the KIV system.

The predicate `safe` ranges over a type of system configurations, each consisting of the `ATM` configuration (`caConf`) and queue, as well as the bank configuration (`cbConf`) and queue. The machine configurations in turn consist of the control state and attributes. The safety predicate holds in a configuration *iff*, whenever `ATM` is in control state `Verified`, the `Bank`'s attribute `wasVerified` has the value 1.

The behaviours of `Bank` and `ATM` are defined in the form of an initial state predicate and a transition predicate. For space reasons we show only one transition:

```
atmTrans-def: atmTrans(atmConf(sa1, c1, p1, t1), in, out, atmConf(sa2, c2, p2, t2))
  ↔ ∃ c : CardId, p : Pin . ...
       ∨ (   sa1 = CardEntered
          ∧ in = msg(userCom, PIN(p)) ∧ out = (msg(atmCompl, PINEnteredCompl) +l [])
          ∧ p2 = p ∧ sa2 = PINEntered ∧ c2 = c1 ∧ t2 = t1)
       ∨ ...; used for: s, ls;
```

The `ATM` transitions from one configuration to another, receiving an input event and sending out a list of messages. Each `ATM` configuration consists of (in that order) the control state, the card id to be verified, the PIN to be verified and the counter for the number of verification attempts. We give the definition of the transition predicate by a disjunction of the conditions of all syntactic transitions, including the control state before, the input event, the output list, variables to be set, the control state after and variables to remain unchanged. Given these machine predicates and a predicate `dist` to encode connectors, we can then define the transition predicate for the overall system:

```
trans-def: trans(conf(ca1, qa1, cb1, qb1), in, out, conf(ca2, qa2, cb2, qb2))
  ↔   dist(out, qa1, qa2, qb1, qb2)
    ∧ (   (atmTrans(ca1, in, out, ca2) ∧ cb2 = cb1 )
        ∨ (bankTrans(cb1, in, out, cb2) ∧ ca2 = ca1)); used for: s, ls;
```

Initially, the queues are empty and the machines are in their initial configurations.

Having thus defined the machines, we turn to verification and define an invariant strong enough to show both its own preservation and our safety property. The essential idea is that we must control when the events allowing us to enter the `Verified` state on the `ATM` or to reset the `wasVerified` attribute of the bank are in the queues.

```
invar-def: invar(conf(ca,qa,cb,qb)) ↔ ∃ x.
    (ctrl(ca) = Idle        ∧ ctrl(cb) = Idle        ∧ qa = empty ∧ qb = empty)
```

```
  ∨ (ctrl(ca) = CardEntered ∧ ctrl(cb) = Idle        ∧ qa = empty ∧ qb = empty)
  ∨ (ctrl(ca) = PINEntered  ∧ ctrl(cb) = Idle        ∧ qa = enq(x, empty) ∧ qb = empty)
  ∨ (ctrl(ca) = Verifying   ∧ ctrl(cb) = Idle        ∧ qa = empty ∧ qb = enq(x, empty))
  ∨ (ctrl(ca) = Verifying   ∧ ctrl(cb) = Verifying   ∧ qa = empty ∧ qb = enq(x, empty))
  ∨ (ctrl(ca) = Verifying   ∧ ctrl(cb) = VeriSuccess ∧
                qa = empty ∧ qb = enq(x, empty) ∧ wasVerified(cb) = 1)
  ∨ (ctrl(ca) = Verifying   ∧ ctrl(cb) = VeriFail    ∧ qa = empty ∧ qb = enq(x,empty))
  ∨ (ctrl(ca) = Verifying   ∧ ctrl(cb) = Idle        ∧
                qa = enq(msg(bankCom, reenterPIN), empty) ∧ qb = empty)
  ∨ (ctrl(ca) = Verifying   ∧ ctrl(cb) = Idle        ∧
                qa = enq(msg(bankCom, verified), empty) ∧ qb = empty ∧ wasVerified(cb) = 1)
  ∨ (ctrl(ca) = Verified    ∧ ctrl(cb) = Idle        ∧
                qa = enq(x, empty) ∧ qb = empty ∧ wasVerified(cb) = 1); used for: s, ls;
```

Note that we can mostly ignore attribute values, as well as all distinctions between queue elements unrelated to our verification task. We can then formulate lemmas to the effect that this invariant does in fact imply the safety property, that it is satisfied in all legal initial configurations and that it is preserved by all transitions. These lemmas are as follows, again limited to one example for the transitions:

```
lemmas
  Safe: invar(g) → safe(g);
  Init: init(g) → invar(g);
  ...
  Trans6: g1 = conf(atmConf(Verifying, c, p, t), qa, cb, qb)
        ∧ qa ≠ empty ∧ top(qa) = msg(atmCom, verified)
        ∧ g2 = conf(atmConf(Verified, c, p, t),
                    enq(msg(atmCompl, VerifiedCompl), deq(qa)), cb, qb)
        ∧ invar(g1) → invar(g2);
```

Formulating separate lemmas for each transition instead of one lemma using the transition predicate helps us avoid a combinatorial explosion in the theorem prover.

Providing our specification to KIV with all definitions marked as simplifier rules and activating the heuristics mode “PL heuristics + structural induction”, each of our lemmas is proved without noticeable delay, i.e., the verification of the invariant is successful and does not pose any difficulty to the prover.

At the moment, verifying other examples with this approach would proceed as follows:

1. formulate the system in UMLStateO and UMLComp
2. translate it (manually) into KIV
3. formulate the desired property
4. formulate the invariant and preservation lemmas
5. let KIV prove the lemmas

We expect that further automation could remove the manual steps 2 and 4, leaving only 1, 3 and 5.

Part III

Conclusions

Chapter 14

Conclusion

Contents

14.1 Summary

The UML was developed to unify existing modelling approaches and has successfully established itself as the main modelling language used in industry. It is the topic of ongoing research to establish formal semantics and tools for rigorous reasoning about UML models and the systems they specify. The work presented in this thesis forms part of one long-term effort [KMR15, KMRG15, KM17, Ros17] to develop such semantics and tools. Through it, we advance this effort significantly by providing concrete input languages and semantics for simple UML State Machines and Composite Structures. Further, we provide tool support for these UML models by implementing our languages in HETS and providing well-behaved translations to CASL, which mediates access to existing general purpose provers, both interactive and fully automatic. We give exemplary models and properties involving typical challenges of inductive reasoning about software systems for which we show the feasibility of fully automatic verification.

We approach the semantics of UML State Machines in two steps: Following a key insight we had in earlier work [Ros17], we first translate via a semantic function from a more direct State Machine description into a modal logic. This modal logic is built to require the presence or absence of sets of transitions in a way that admits the semantics-preserving translations of Institution Theory (i.e., signature morphisms and institution comorphisms). Once the semantic function has brought us into the realm of Institution Theory, we can then gain a semantically sound relationship with other formalisms by means of comorphisms.

We develop our State Machine language in two steps. The first only involves State Machines without output events, the second involves output events and adds ports in a somewhat transparent way. These extensions allow us to compose State Machines using an institution of Composite Structures, which we also introduce. Our two State Machine institutions and our Composite Structure institution are equipped with comorphisms to CASL, which can be composed with the

numerous comorphisms out of CASL to other logics, allowing us to borrow [CM97] their proof systems and reasoning tools.

To sum up, our theoretical results are three new institutions for modelling sublanguages of the UML, each with a comorphism to CASL. Our practical contributions are

– the development of concrete input languages for these institutions and a prototypical implementation for these languages and for our comorphisms in the Heterogeneous Toolset HETS and

– the verification, as a proof of concept, of two concrete example systems specified in UML, supported by our theory and tools.

14.2 Future Work

There are several immediate continuations of our work we can suggest. First, we suggest to create full automation for the generation of the auxiliary lemmas we needed to establish invariants in our examples. Next, it would be useful to link the data universe of EDHMLO to UML Class Diagrams (possibly following [JKMR12]) and make types from the Class Diagrams available inside the implemented State Machines.

A larger project which would complete our automation, but also bring benefits beyond the world of UML, would be the addition of KIV to HETS. In our practical verification attempts, KIV's direct support for inductive data types and for giving operational meanings to axioms or theorems made results feasible which were simply out of reach using HETS's existing fully automatic provers. Not all the relevant constructs of KIV have direct correlates in CASL. In our experiments, our use of these inductive and operational KIV features was limited to defining a well-founded relation on the natural numbers and marking all axioms without distinction as rewrite rules. In general, good and fully automatic proof support might require one or more comorphisms using heuristics to recognise which of KIV's proof support constructs should be used where. Further, although there seems to be no reason to think KIV would not be an institution, to our knowledge this has never been formally established. To sum up, integrating KIV into HETS would be a larger project, possibly a PhD project in itself, which could greatly improve HETS's fully automatic capabilities for inductive reasoning, which is a central ingredient to the verification of models and programs.

Casting the net wider in the area of UML verification, it would be worthwhile to integrate UML Interaction Diagrams into HETS and carry out experiments on how they can be related to State Machines and Composite Structures. Further, we suggest that there should be a systematic, experimental and theoretical study of notions of refinement between different UML models in HETS and Institution Theory. The signature morphisms (for models of the same kind) and the comorphisms (between different kinds of models) would be the obvious starting points in the search for such notions.

Moreover, it would be worthwhile to connect to other formalisms for concurrent programming. Two such formalisms which are already available in the institutional world are the process algebra CSP and the (non-UML) State Machine based language Event-B. For CSP, an integration with CASL was developed in the form of `CSP-CASL` [Rog06], an implementation of which

[Gim08, O'R12] has been integrated into upstream HETS. Likewise, in the case of Event-B, there is a formalisation as an institution [FMP17]. Event-B is a descendant of the successful B-method, which has been used, among other applications, to develop the verified control software for the driverless Paris metro lines [BG00]. The Event-B institution of [FMP17] has also been implemented in a fork[1] of HETS. The fork features an integration of the Rodin tool for analysing Event-B models [FMP17]. Unfortunately, it seems that this fork was never merged into upstream HETS and has received no updates since 2015. It would be an opportunity for future work to integrate Event-B into upstream HETS and perform a mixed case study involving, e.g., industry-supplied UML Composite Structure Diagrams and State Machines, a desired property expressed in CSP, and a development process by refinement in and code-generation from Event-B, all the while using HETS to relate the different languages.

Finally, we would propose future work on the trustworthiness of HETS itself. This is not only a moving target, as new logics get added to HETS, but it is also a subtle challenge: A minimal trusted core in the tradition of LCF [Mil79, Har96] can easily guarantee that a proof does in fact prove something. However, it is in general difficult to ensure that what has been proved is the intended property. In the case of a heterogeneous specification, this problem becomes even more difficult, as it involves the semantics of several languages and the translations between these. Making all of these part of a trusted core can hardly be acceptable. One way forward could be to create a high-trust code path connecting the proof rules on development graphs, as well as basic CASL specifications (or a subset thereof) to the core languages of one or a few provers. Trust in this code path could be established by a mixture of semantic modelling, formally proved theorems, and testing. Trust in other provers, where they output proof terms, can be replaced by reconstruction of their proof terms in one of the trusted core languages.

It would then be possible to model a system in a less trustworthy language of the HETS ecosystem, and express the main theorems over this system in CASL (or its trusted subset), like we did in our verification examples. Auxiliary lemmas could then be phrased in any convenient language supported by HETS. A lemma can then be translated into the language of a trusted prover, can be proved in that form, and used to prove the ultimately desired theorems. The correctness of the proof of a theorem would then depend on the formal correctness of the proofs of the translated lemmas within the trusted core, but not on the formal or intuitive correctness of the lemmas as originally phrased, or of their translation procedure into the core.

We are, of course, not the first to propose a verification of the logics and translations included in HETS. In particular, the work we propose, in its formal aspects, will find a strong basis in the infrastructure and the logic atlas created by the LATIN project [CHK+10].

[1] The Event-B fork of HETS is available at `https://github.com/mariefarrell/Hets`.

Bibliography

[ALSU07] Alfred V. Aho, Monika S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison/Wesley, Boston, 2nd edition, 2007.

[BBK+04] Michael Balser, Simon Bäumler, Alexander Knapp, Wolfgang Reif, and Andreas Thums. Interactive Verification of UML State Machines. In Jim Davies, Wolfram Schulte, and Mike Barnett, editors, *Proc. 6th Intl. Conf. Formal Engineering Methods (ICFEM 2004)*, volume 3308 of *Lect. Notes Comp. Sci.*, 2004.

[BG00] J. L. Boulanger and M. Gallardo. Validation and verification of meteor safety software. *WIT Transactions on The Built Environment*, 50, 2000.

[BM03] Michel Bidoit and Peter D. Mosses. *CASL User Manual: Introduction to Using the Common Algebraic Specification Language*, volume 2900. Springer, 2003.

[BRJ98] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, October 1998.

[BS95] Patrick Blackburn and Jerry Seligman. Hybrid languages. *Journal of Logic, Language and Information*, 4(3):251–272, 1995.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on logic of programs*, pages 52–71. Springer, 1981.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[CHK+10] Mihai Codescu, Fulya Horozal, Michael Kohlhase, Till Mossakowski, Florian Rabe, and Kristina Sojakova. Towards logical frameworks in the heterogeneous tool set hets. In *International Workshop on Algebraic Development Techniques*, pages 139–159. Springer, 2010.

[CM97] Maura Cerioli and José Meseguer. May I borrow your logic? (transporting logical structures along maps). *Theoretical Computer Science*, 173(2):311–347, 1997.

[DM03] Francisco Durán and José Meseguer. Structured theories and institutions. *Theoretical Computer Science*, 309(1-3):357–380, 2003.

[DM16] Razvan Diaconescu and Alexandre Madeira. Encoding Hybridized Institutions into First-order Logic. *Math. Struct. Comp. Sci.*, 26(5):745–788, 2016.

[DY83] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[EPS+15] Gidon Ernst, Jörg Pfähler, Gerhard Schellhorn, Dominik Haneberg, and Wolfgang Reif. KIV: overview and VerifyThis competition. *International Journal on Software Tools for Technology Transfer*, 17(6):677 – 694, 2015.

[FL08] Andrew Forward and Timothy C. Lethbridge. Problems and opportunities for model-centric versus code-centric software development: a survey of software professionals. In *Proceedings of the 2008 international workshop on Models in software engineering*, pages 27–32, 2008.

[FMP17] Marie Farrell, Rosemary Monahan, and James F Power. Combining Event-B and CSP: An institution theoretic approach to interoperability. In *International Conference on Formal Engineering Methods*, pages 140–156. Springer, 2017.

[GB83] Joseph A. Goguen and Rod M. Burstall. Introducing institutions. In *Workshop on Logic of Programs*, pages 221–256. Springer, 1983.

[GB92] Joseph A Goguen and Rod M Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM (JACM)*, 39(1):95–146, 1992.

[Gim08] Andy Gimblett. Tool support for CSP-CASL. Master's thesis, Swansea University, 2008.

[GR02] Joseph Goguen and Grigore Roşu. Institution morphisms. *Formal aspects of computing*, 13(3):274–307, 2002.

[Grö10] Hans Grönniger. *Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten*. PhD thesis, RWTH Aachen, 2010.

[Har96] John Harrison. HOL light: A tutorial introduction. In *International Conference on Formal Methods in Computer-Aided Design*, pages 265–269. Springer, 1996.

[HMK19] Rolf Hennicker, Alexandre Madeira, and Alexander Knapp. A Hybrid Dynamic Logic for Event/Data-Based Systems. In Reiner Hähnle and Wil M. P. van der Aalst, editors, *Proc. 22nd Intl. Conf. Fundamental Approaches to Software Engineering*, volume 11424 of *Lect. Notes Comp. Sci.*, pages 79–97. Springer, 2019.

[JKMR12] Phillip James, Alexander Knapp, Till Mossakowski, and Markus Roggenbach. Designing Domain Specific Languages — A Craftsman's Approach for the Railway Domain Using CASL. In Narciso Martí-Oliet and Miguel Palomino, editors, *Rev. Sel. Papers 21st Intl. Ws. Recent Trends in Algebraic Development Techniques (WADT 2012)*, volume 7841 of *Lect. Notes Comp. Sci.*, pages 178–194. Springer, 2012.

[JKMR13] Phillip James, Alexander Knapp, Till Mossakowski, and Markus Roggenbach. Designing domain specific languages–a craftsman's approach for the railway domain using CASL. In *International Workshop on Algebraic Development Techniques*, pages 178–194. Springer, 2013.

[Kah87] Gilles Kahn. Natural semantics. In *Annual symposium on theoretical aspects of computer science*, pages 22–39. Springer, 1987.

[KCL07] Il Jung Kim, Eun Young Choi, and Dong Hoon Lee. Secure mobile RFID system against privacy and security problems. In *3rd Intl. Ws. Security, Privacy and Trust in Pervasive and Ubiquitous Computing (SecPerU 2007)*, pages 67–72, 2007.

[KFdB+05] Marcel Kyas, Harald Fecher, Frank S. de Boer, Joost Jacob, Jozef Hooman, Mark van der Zwaag, Tamarah Arons, and Hillel Kugler. Formalizing UML Models and OCL Constraints in PVS. In Gerald Lüttgen and Michael Mendler, editors, *Proc. Ws. Semantic Foundations of Engineering Design Languages (SFEDL 2004)*, volume 115 of *Electr. Notes Theo. Comp. Sci.*, 2005.

[KM17] Alexander Knapp and Till Mossakowski. UML Interactions Meet State Machines — An Institutional Approach. In Filippo Bonchi and Barbara König, editors, *Proc. 7th Intl. Conf. Algebra and Coalgebra in Computer Science (CALCO 2017)*, volume 72 of *LIPIcs*, pages 15:1–15:15, 2017.

[KMR15] Alexander Knapp, Till Mossakowski, and Markus Roggenbach. Towards an Institutional Framework for Heterogeneous Formal Development in UML — A Position Paper. In Rocco De Nicola and Rolf Hennicker, editors, *Software, Services, and Systems — Essays Dedicated to Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering*, volume 8950 of *Lect. Notes Comp. Sci.*, pages 215–230. Springer, 2015.

[KMRG15] Alexander Knapp, Till Mossakowski, Markus Roggenbach, and Martin Glauer. An Institution for Simple UML State Machines. In Alexander Egyed and Ina Schaefer, editors, *Proc. 18th Intl. Conf. Fundamental Approaches to Software Engineering (FASE 2015)*, volume 9033 of *Lect. Notes Comp. Sci.*, pages 3–18. Springer, 2015.

[Lam06] Leigh Lambert. Modal logic in computer science. 2006.

[LM01] Daan Leijen and Erik Meijer. Parsec: A practical parser library. *Electronic Notes in Theoretical Computer Science*, 41(1):1–20, 2001.

[M+10] Simon Marlow et al. Haskell 2010 language report. *Available on: https://www. haskell. org/onlinereport/haskell2010*, 2010.

[Mad13] Alexandre Madeira. *Foundations and Techniques for Software Reconfigurability*. PhD thesis, Universidade do Minho, 2013.

[MBHM16] Alexandre Madeira, Luis S. Barbosa, Rolf Hennicker, and Manuel A. Martins. Dynamic Logic with Binders and Its Application to the Development of Reactive Systems. In *Proc. 13th Intl. Coll. Theoretical Aspects of Computing*, volume 9965 of *Lect. Notes Comp. Sci.*, pages 422–440. Springer, 2016.

[Mes89] José Meseguer. General logics. In *Studies in Logic and the Foundations of Mathematics*, volume 129, pages 275–329. Elsevier, 1989.

[Mil79] Robin Milner. LCF: A way of doing proofs with a machine. In *International Symposium on Mathematical Foundations of Computer Science*, pages 146–159. Springer, 1979.

[MMCL14] Till Mossakowski, Christian Maeder, Mihai Codescu, and Dominik Lucke. Hets user guide-version 0.99, 2014.

[MML07a] Till Mossakowski, Christian Maeder, and Klaus Lüttich. The heterogeneous tool set, Hets. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 519–522. Springer, 2007.

[MML07b] Till Mossakowski, Christian Maeder, and Klaus Lüttich. The Heterogeneous Tool Set. In Orna Grumberg and Michael Huth, editors, *Proc. 13th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)*, volume 4424 of *Lect. Notes Comp. Sci.*, pages 519–522. Springer, 2007.

[Mos97] Peter D. Mosses. Cofi: The common framework initiative for algebraic specification and development. In *Colloquium on Trees in Algebra and Programming*, pages 115–137. Springer, 1997.

[Mos02] Till Mossakowski. Comorphism-based grothendieck logics. In *International Symposium on Mathematical Foundations of Computer Science*, pages 593–604. Springer, 2002.

[Mos05] Till Mossakowski. *Heterogeneous Specification and the Heterogeneous Tool Set*. Habilitation thesis, University of Bremen, 2005.

[Mos16] Till Mossakowski. The distributed ontology, model and specification language–DOL. In *Rev. Sel. Papers 23rd Intl. Ws. Algebraic Development Techniques (WADT 2016)*, volume 10644 of *Lect. Notes Comp. Sci.*, pages 5–10. Springer, 2016.

[MSLF14] Franco Mazzanti, Giorgio Oronzo Spagnolo, Simone Della Longa, and Alessio Ferrari. Deadlock avoidance in train scheduling: a model checking approach. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 109–123. Springer, 2014.

[NGYJ21] Faranak Nejati, Abdul Azim Abd Ghani, Ng Keng Yap, and Azmi Bin Jafaar. Handling state space explosion in component-based software verification: A review. *IEEE Access*, 9:77526–77544, 2021.

[Obj17] Object Management Group. Unified Modeling Language. Standard formal/17-12-05, OMG, 2017.

[Obj19] Object Management Group. Precise Semantics of UML Composite Structures. Standard formal/2019-02-01, OMG, 2019.

[OE20] Mert Ozkaya and Ferhat Erata. A survey on the practical use of UML for different software architecture viewpoints. *Information and Software Technology*, 121:106275, 2020.

[O'R12] Liam O'Reilly. *Structured Specification with Processes and Data*. PhD thesis, Swansea University, 2012.

[pla] Drawing UML with PlantUML – PlantUML language reference guide (version 1.2021.2).

[Plo81] Gordon D Plotkin. *A structural approach to operational semantics*. Aarhus university, 1981.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on programming*, pages 337–351. Springer, 1982.

[RBKR20] Tobias Rosenberger, Saddek Bensalem, Alexander Knapp, and Markus Roggenbach. Institution-based Encoding and Verification of Simple UML State Machines in CASL/SPASS. In Markus Roggenbach, editor, *Rev. Sel. Papers 25th Intl. Ws. Recent Trends in Algebraic Development Techniques (WADT 2020)*, volume 12669 of *Lect. Notes Comp. Sci.*, pages 120–141. Springer, 2020.

[RHQH$^+$17] Gregorio Robles, Truong Ho-Quang, Regina Hebig, Michel RV Chaudron, and Miguel Angel Fernandez. An extensive dataset of UML models in GitHub. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 519–522. IEEE, 2017.

[RKR22] Tobias Rosenberger, Alexander Knapp, and Markus Roggenbach. An institutional approach to communicating UML State Machines. In *25th Intl. Conf. Fundamental Approaches to Software Engineering (FASE 2022)*, pages 205–224. Springer, 2022.

[RMS03] Markus Roggenbach, Till Mossakowski, and Lutz Schröder. CASL reference manual. 2003.

[Rog06] Markus Roggenbach. CSP-CASL – a new integration of process algebra and algebraic specification. *Theoretical Computer Science*, 354(1):42–71, 2006.

[Ros17] Tobias Rosenberger. Relating UML state machines and interactions in an institutional framework. Master's thesis, Elite Graduate Program Software Engineering (Universität Augsburg, Ludwig-Maximilians-Universität München, Technische Universität München), 2017.

[RSN22] Markus Roggenbach, Siraj Ahmed Shaikh, and Hoang Nga Nguyen. Formal verification of security protocols. In Markus Roggenbach, Antonio Cerone, Bernd-Holger Schlingloff, Gerardo Schneider, and Siraj Ahmed Shaikh, editors, *Formal Methods for Software Engineering – Languages, Methods, Application Domains*, pages 393–449. Springer, 2022.

[Sco82] Dana S. Scott. Domains for denotational semantics. In *International Colloquium on Automata, Languages, and Programming*, pages 577–610. Springer, 1982.

[SWK+10] Mathias Soeken, Robert Wille, Mirco Kuhlmann, Martin Gogolla, and Rolf Drechsler. Verifying UML/OCL models using boolean satisfiability. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 1341–1344. IEEE, 2010.

[tBGM15] Maurice H. ter Beek, Stefania Gnesi, and Franco Mazzanti. From EU projects to a family of model checkers. In *Software, Services, and Systems*, pages 312–328. Springer, 2015.

[TTR+13] Marco Torchiano, Federico Tomassetti, Filippo Ricca, Alessandro Tiso, and Gianna Reggio. Relevance, benefits, and problems of software modelling and model driven techniques—a survey in the italian industry. *Journal of Systems and Software*, 86(8):2110–2126, 2013.

[VPMY04] Valentin Goranko, Patrick Blackburn, Maarten de Rijke, and Yde Venema. Modal logic, Cambridge Tracts in Theoretical Computer Science. 53, 2004.

[WBCW20] Andreas Wortmann, Olivier Barais, Benoit Combemale, and Manuel Wimmer. Modeling languages in industry 4.0: an extended systematic mapping study. *Software and Systems Modeling*, 19(1):67–94, 2020.

[WDF+09] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS Version 3.5. In Renate A. Schmidt, editor, *Proc. 22nd Intl. Conf. Automated Deduction*, volume 5663 of *Lect. Notes Comp. Sci.*, pages 140–145. Springer, 2009.

Appendix A

Language Definitions for UMLSTATE

A.1 Lexical Primitives

The following lexical primitives are available. Their syntax is that of ordinary (non-mixfix) CASL identifiers or of (unsigned decimal integer) numerals.

```
  VAR_NAME ::= Token
EVENT_NAME ::= Token
     STATE ::= Token
   NAT_LIT ::= Int
```

A.2 Abstract Syntax

With these lexical primitives in place, we can now define the abstract syntax of UMLSTATE. We stay close to the BNF-like notation usual in CASL-related work, our main departure being the explicit parentheses for rules including terminal symbols: `terminal` ⟨`NONTERMINALS`⟩

This choice emphasises the tree structure which the grammar induces on each word of the language, which simplifies our description of the static semantics.

Each rule of the abstract syntax expands a non-terminal symbol, written to the left of ::=. The expansion, written on the right may contain a leading terminal symbol, written in lower case. If a terminal symbol is present, the rest of the expansion is parenthesised as shown above. The remaining symbols are non-terminals, where the Kleene plus can be added to indicate the possibility of repetition. Both non-terminals and parts of terminals can be underlined to mark them as optional.

Given these conventions, the following grammar is then an abstract description of parse trees for UMLSTATE basic specifications, which can be embedded in HETS heterogeneous specifications. The terminal symbol at the root is `BASIC_SPEC`.

`BASIC_SPEC` ::= `basic-spec` ⟨`BASIC_ITEMS`$^+$ ⟩

A basic specification consists of a number of items, each of which introduces variables or events into the signature, or describes a (possibly initial) transition the machine should contain.

```
BASIC_ITEMS  ::=  sig-its⟨SIG_ITEMS⟩
               |  var-its⟨VAR_ITEMS⟩
               |  evt-its⟨EVENT_ITEMS⟩
               |  sen-its⟨SEN_ITEMS⟩
  VAR_ITEMS  ::=  VAR_NAME+
  SEN_ITEMS  ::=  trans-it⟨TRANS_ITEM⟩
               |  init⟨STATE GUARD⟩
EVENT_ITEMS  ::=  evt-it⟨EVENT_ITEM+ ⟩
 EVENT_ITEM  ::=  EVENT_NAME VAR_NAME+ VAR_NAME
  SIG_ITEMS  ::=  st-it⟨STATE_ITEM+⟩
 STATE_ITEM  ::=  STATE
 TRANS_ITEM  ::=  STATE STATE TRANS_LABEL
TRANS_LABEL  ::=  TRIGGER GUARD ACTIONS
    TRIGGER  ::=  EVENT_ITEM
      GUARD  ::=  FORMULA
    ACTIONS  ::=  ACTION+
```

The following terms, actions and data formulae are available in UMLSTATE as we implemented it. Note that these are specific to the natural numbers as our data universe, whereas our institution as formalised is parametric in the data universe used. Data formulae are a subset of CASL.

```
     TERM  ::=  var-term ⟨VAR_NAME⟩
             |  lit-term ⟨NAT_LIT⟩
             |  plus-term ⟨TERM TERM⟩
             |  minus-term⟨TERM TERM⟩
             |  mult-term ⟨TERM TERM⟩
             |  div-term ⟨TERM TERM⟩
  FORMULA  ::=  comp-form⟨TERM COMP_OP TERM⟩
             |  true-form | false-form
             |  not-form⟨FORMULA⟩
             |  and-form⟨FORMULA FORMULA⟩
             |  or-form⟨FORMULA FORMULA⟩
             |  impl-form⟨FORMULA FORMULA⟩
             |  if-form⟨FORMULA FORMULA⟩
             |  equiv-form⟨FORMULA FORMULA⟩
  COMP_OP  ::=  less-op | less-eq-op | eq-op | greater-eq-op | greater-op
   ACTION  ::=  VAR_NAME TERM
```

A.3 Concrete Syntax

The grammar for the concrete syntax follows conventions similar to those for the abstract syntax. We make the connection between the two grammars by using the same non-terminal symbols. Terminal symbols can now be anything that is not a non-terminal or meta-symbol, and are freely mixed with the non-terminals. Repetition now indicates a separator to be used (comma or semicolon) and is given in the form `NON_TERM;...;NON_TERM` or `NON_TERM,...,NON_TERM`.

As in the case of the abstract syntax, we start with the rules for basic specifications and the various items they can consist of.

```
  BASIC_SPEC  ::=  BASIC_ITEMS;...;BASIC_ITEMS
 BASIC_ITEMS  ::=  SIG_ITEMS
             |     VAR_ITEMS
             |     SEN_ITEMS
             |     EVENT_ITEMS
   VAR_ITEMS  ::=  vars VAR_NAME,...,VAR_NAME
   SEN_ITEMS  ::=  TRANS_ITEM
             |     init STATE : GUARD
EVENT_ITEMS   ::=  events EVENT_ITEM,...,EVENT_ITEM
 EVENT_ITEM   ::=  EVENT_NAME (VAR_NAME,...,VAR_NAME)
  SIG_ITEMS   ::=  states STATE_ITEM,...,STATE_ITEM
 STATE_ITEM   ::=  STATE
 TRANS_ITEM   ::=  trans STATE - - > STATE : TRANS_LABEL
TRANS_LABEL   ::=  TRIGGER GUARD ACTIONS
    TRIGGER   ::=  EVENT_ITEM
      GUARD   ::=  [FORMULA]
    ACTIONS   ::=  /{ACTION;...;ACTION}
```

The following rules are once more specific to our choice of data universe:

```
  ACTION  ::=  VAR_NAME := TERM
 FORMULA  ::=  TERM COMP_OP TERM
          |    true | false
          |    not FORMULA
          |    FORMULA /\ FORMULA
          |    FORMULA \/ FORMULA
          |    FORMULA => FORMULA
          |    FORMULA if FORMULA
          |    FORMULA <=> FORMULA
          |    (FORMULA)
    TERM  ::=  VAR_NAME
          |    NAT_LIT
          |    TERM + TERM
          |    TERM - TERM
          |    TERM * TERM
          |    TERM / TERM
          |    (TERM)
COMP_OP   ::=  < | <= | == | >= | >
```

A.4 Static Semantics

With the abstract and concrete syntax in place, we can now turn to the description of the static semantics of UMLSTATE, which we will describe in the style of natural semantics ([Plo81, Kah87]).

Our main judgements are of the form $\Sigma \vdash w \triangleright (\Delta, \Psi)$, meaning that over a signature Σ a word w is well-formed, and produces a signature extension Δ and a set Φ of transition sentences (i.e., `SEN_ITEM`s). A signature is a triple of sets, the first containing states (`STATE_ITEM`s), the second attributes (`VAR_ITEM`s), the third events (the set is a function from name-arity pairs to `EVENT_ITEM`s). A signature extension is given by the same data. We make use of unions of signatures and signature extensions, which we define componentwise. We assume w to be a word over our abstract grammar. It is useful to think of these words as trees. in which each terminal symbol from the abstract grammar is an inner node and its parentheses enclose a subtree. This way, non-terminals whose defining rule involves no terminal symbol do not form sub-trees of their own and need not here be treated by their own semantic rules. Where there is no derivation tree to infer the well-formedness of a word in a given context, we consider it ill-formed, i.e., well-formedness is defined inductively.

We start with four rules which merely traverse subtrees and take the union of all the signature extensions and sentences produced. Earlier definitions are in scope for later ones. These rules are identical, except for the terminal symbol of the interpreted word.

$$\frac{\Sigma \cup \bigcup_{i=1}^{0} \Delta_i \vdash it_1 \triangleright (\Delta_1, \Psi_1) \qquad \cdots \qquad \Sigma \cup \bigcup_{i=1}^{n-1} \Delta_i \vdash it_n \triangleright (\Delta_n, \Psi_n)}{\Sigma \vdash \texttt{basic-spec}\,\langle it_1 \ldots it_n \rangle \triangleright (\bigcup_{i=1}^{n} \Delta_i, \bigcup_{i=1}^{n} \Psi_i)}$$

$$\frac{\Sigma \cup \bigcup_{i=1}^{0} \Delta_i \vdash it_1 \triangleright (\Delta_1, \Psi_1) \qquad \cdots \qquad \Sigma \cup \bigcup_{i=1}^{n-1} \Delta_i \vdash it_n \triangleright (\Delta_n, \Psi_n)}{\Sigma \vdash \texttt{var-its}\,\langle it_1 \ldots it_n \rangle \triangleright (\bigcup_{i=1}^{n} \Delta_i, \bigcup_{i=1}^{n} \Psi_i)}$$

$$\frac{\Sigma \cup \bigcup_{i=1}^{0} \Delta_i \vdash it_1 \triangleright (\Delta_1, \Psi_1) \qquad \cdots \qquad \Sigma \cup \bigcup_{i=1}^{n-1} \Delta_i \vdash it_n \triangleright (\Delta_n, \Psi_n)}{\Sigma \vdash \texttt{sen-its}\,\langle it_1 \ldots it_n \rangle \triangleright (\bigcup_{i=1}^{n} \Delta_i, \bigcup_{i=1}^{n} \Psi_i)}$$

$$\frac{\Sigma \cup \bigcup_{i=1}^{0} \Delta_i \vdash it_1 \triangleright (\Delta_1, \Psi_1) \qquad \cdots \qquad \Sigma \cup \bigcup_{i=1}^{n-1} \Delta_i \vdash it_n \triangleright (\Delta_n, \Psi_n)}{\Sigma \vdash \texttt{evt-its}\,\langle it_1 \ldots it_n \rangle \triangleright (\bigcup_{i=1}^{n} \Delta_i, \bigcup_{i=1}^{n} \Psi_i)}$$

An attribute can be added if it is not already in the signature. This produces a signature extension containing only the new attribute.

$$\frac{v \notin Attr}{(State, Attr, Evt) \vdash \texttt{var-it}\,\langle v \rangle \triangleright ((\emptyset, \{v\}, \emptyset), \Psi)}$$

A state can be introduced if it is not already in the signature. The resulting signature extension contains only the new state.

$$\frac{s \notin State}{(State, Attr, Evt) \vdash \texttt{st-it}\,\langle s\rangle \triangleright ((\{s\},\emptyset,\emptyset),\emptyset)}$$

A sentence for an initial transition can be introduced if the target states and all variables of the guard are in the signature. The sentence is given as an added sentence.

$$\frac{s \in State \qquad vars(g) \subseteq Attr}{(State, Attr, Evt) \vdash \texttt{init}\,\langle s\ g\rangle \triangleright (\Delta, \{(s,g)\})}$$

The definition of a new event is valid if no event with the same name and arity is already contained in the signature. The signature extension contains just a mapping from the name and arity of the new event to the word defining it.

$$\frac{(name, |vars|) \notin dom(Evt)}{(State, Attr, Evt) \vdash \texttt{evt-it}\,\langle name\ vars\rangle \triangleright ((\emptyset,\emptyset,\{(name,|vars|) \mapsto \texttt{evt-it}\,\langle name\ vars\rangle\},\emptyset)}$$

A transition item is valid if its source and target state are defined, the trigger is defined up to renaming of parameters and the guard and actions are valid. We define guard and action validity by separate judgement forms. Their definitions amount to saying that a guard is valid if all its variables are either defined attributes or trigger parameters, and an action is valid if it assigns a valid term to a known attribute. Term validity follows the same rule as guard validity.

$$\frac{\{s_1,s_2\} \subseteq State \qquad (Attr,e) \vdash g\ \texttt{guard} \qquad (Attr,Evt,e) \vdash a\ \texttt{actions}}{(State, Attr, Evt) \vdash \texttt{trans-it}\,\langle s_1\ s_2\ e\ g\ a\rangle \triangleright ((\emptyset,\emptyset,\emptyset),\{\texttt{trans-it}\,\langle s_1\ s_2\ e\ g\ a\rangle\})}$$

$$\frac{vars(g) \subseteq Attr \cup evars}{(Attr, \texttt{evt-it}\,\langle ename\ evars\rangle) \vdash g\ \texttt{guard}}$$

$$\frac{(ename, |evar|) \in dom(Evt) \qquad \bigcup_{i=1}^{n}\{v_i\} \subseteq Attr \qquad \bigcup_{i=1}^{n} vars(t_i) \subseteq Attr \cup evars}{(Attr, Evt, \texttt{event-item}\,\langle ename\ evars\rangle) \vdash \texttt{act-it}\,\langle v_1\ t_1\rangle \ldots \texttt{act-it}\,\langle v_n\ t_n\rangle\ \texttt{actions}}$$

Appendix B

Diagrams for the Security Protocol Example

Here we show the remaining diagrams for the State Machine example. Note that the nondeterministic choice expression used by the Dolev-Yao intruder is not directly representable in UMLState or UMLStateO. We can could, however, express it in Casl and still link to it via UMLComp.

The reader will here observe a downside of PlantUML, namely, that it is not good at laying out self-loops.

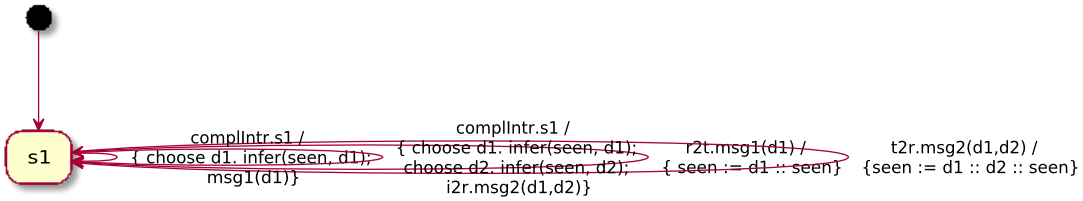


Figure B.1: State Machine for the *DolevYaoIntruder* from the security protocol.

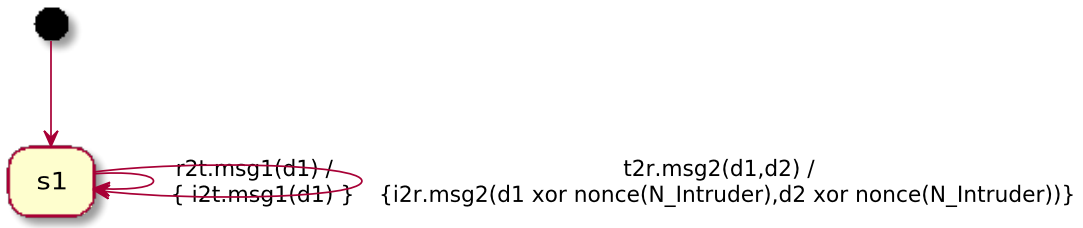


Figure B.2: State Machine for the *SimpleIntruder* from the security protocol.

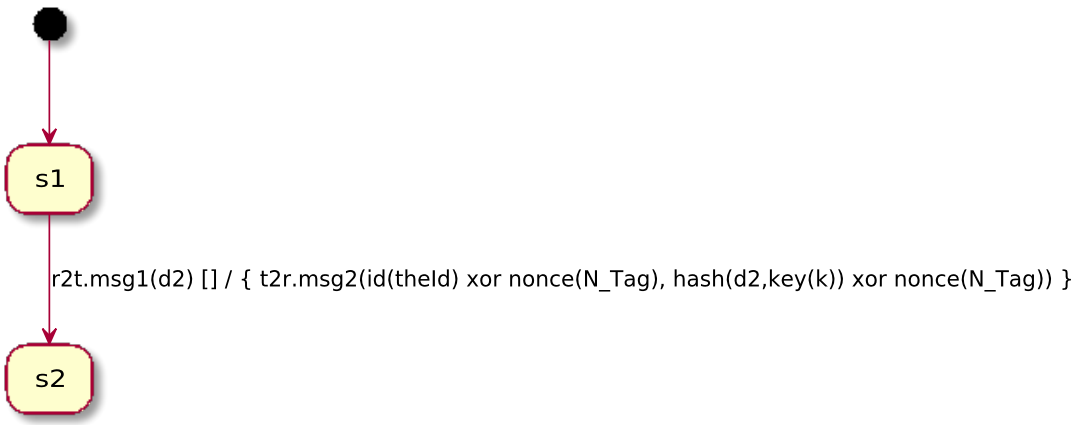


Figure B.3: State Machine for the *Tag* from the security protocol

Appendix C

A Heterogeneous Specification for the Counter example

```
%prec({__+__}<{__*__})%
%right_assoc(__+__, __*__)%
%number __@@__

logic CASL
spec Nat =
  type Nat ::= 0 | suc (Nat)
  op __ + __ : Nat * Nat -> Nat;
     __ * __ : Nat * Nat -> Nat;
  pred __<=__, __ < __, __>__, __>=__, __==__ : Nat * Nat

  forall n,m: Nat
  . 0 + n = n
  . suc(n) + m = suc(n+m)

  . 0 * n = 0
  . suc(n) * m = m + (n * m)

  . 0 <= n
  . not suc(n) <= 0
  . suc(m) <=  suc(n) <=> m <= n

  . m >=n <=> n <= m
  . m < n <=> m <= n /\ not m=n
  . m > n <=> n < m

  . m == n <=> m = n
```

```
then %def
  %% Operations to represent natural numbers with digits:
  ops 1: Nat = suc(0);
      2: Nat = suc(1);
      3: Nat = suc(2);
      4: Nat = suc(3);
      5: Nat = suc(4);
      6: Nat = suc(5);
      7: Nat = suc(6);
      8: Nat = suc(7);
      9: Nat = suc(8);
      __@@__(m: Nat; n: Nat): Nat = m * suc(9) + n %(decimal_def)%

then
  op __ + __ : Nat * Nat -> Nat, assoc, comm, unit 0;
     __ * __ : Nat * Nat -> Nat, assoc, comm, unit 1;

  forall r,s,t: Nat
  . (r + s) * t = r * t + s * t
  . t * (r + s) = t * r + t * s
end

spec GenerateSet [sort Elem] = %mono
  generated type Set[Elem] ::= {} | __+__ (Set[Elem];Elem)
  pred __eps__: Elem * Set[Elem]
  forall x,y: Elem; M,N: Set[Elem]
  . not x eps {}                                    %(elemOf_empty_Set)%
  . x eps M+y <=> x=y \/ x eps M
    %(elemOf_NonEmpty_Set)%
  . M = N <=> (forall x: Elem . x eps M <=> x eps N)  %(equality_Set)%

end

spec Set [sort Elem] given Nat = %mono
  GenerateSet [sort Elem]
then %def
  preds  isNonEmpty:       Set[Elem];
         __ isSubsetOf __: Set[Elem] * Set[Elem]
  ops    {__}: Elem -> Set[Elem];
         #__: Set[Elem] -> Nat;
         __ + __ : Elem * Set[Elem] -> Set[Elem];
         __ - __ : Set[Elem] * Elem -> Set[Elem];
         __ intersection __,
         __ union__,
```

```
        __ - __,
        __ symDiff __:  Set[Elem] * Set[Elem] ->   Set[Elem];


logic UMLState
spec Counter =
  var cnt;

  event inc(k);
  event reset;

  states s1,s2;

  init s1 : [cnt == 0];
  trans s1 --> s1 : inc(k) [cnt+k<4]/{cnt:=cnt+k};
  trans s1 --> s2 : inc(k) [cnt+k==4]/{cnt:=cnt+k};
  trans s2 --> s1 : reset [cnt==4]/{cnt:=0}

end

logic CASL
spec CounterCASL = Counter with translation UMLState2CASL
spec EvtNameSet = Set[sort EvtName] with Set[EvtName] |-> EvtNameSet

spec Trans =
  Nat
then
  CounterCASL
then
  EvtNameSet
then
  forall e:Evt; k,l:Nat
  . (e=evt_reset) \/ (exists k:Nat. e=evt_inc(k))
  . not (evt_reset=evt_inc(k))
  . evt_inc(k)=evt_inc(l) <=> k=l

  type Conf ::= conf(ctrl:Ctrl; cnt:Nat)
  forall g:Conf. exists c:Ctrl; k:Nat
  . g=conf(c,k)
  forall c,d:Ctrl; k,l:Nat
  . conf(c,k)=conf(d,l) <=> c=d /\ k=l
end

spec Prove =
```

```
  Trans
then
  op s1, s2 : Ctrl
  axiom not(s1=s2)
then
  op allEvts : EvtNameSet = evtName_inc + evtName_reset + {}
  pred reachable1(g:Conf) <=>
    init(g) \/ exists g0:Conf; e:Evt
                . reachable1(g0) /\ trans(g0,e,g)

  axiom exists g:Conf. init(g)

   forall g1,g2:Conf. init(g1) /\ init(g2) => ctrl(g1) = ctrl(g2)
  pred invar(g:Conf) <=>
       (ctrl(g)=s1 /\ cnt(g)<=4)
    \/ (ctrl(g)=s2 /\ cnt(g)<=4)

  %% induction scheme for "reachable" predicate, instantiated for invar
  forall es:EvtNameSet
  . (
      forall g:Conf
      . init(g) => invar(g)
    ) /\ (
      forall g,g':Conf; e:Evt
      . (reachable2(es,g) => invar(g)) /\ reachable2(es,g) /\
    trans(g,e,g') => invar(g')
    ) => (
      forall g:Conf
      . reachable2(es,g) => invar(g)
    ) %(InvarIsReachableInd)%

then %implies

  %% case distinction lemmas, can be generated algorithmically
  forall g,g':Conf; e:Evt; k:Nat
  . init(g) => invar(g) %(InvarInit)%
  . (reachable2(allEvts,g) => invar(g)) /\ reachable2(allEvts,g) /\
    trans(g,e,g') /\ e=evt_reset  => invar(g') %(InvarReset)%
  . (reachable2(allEvts,g) => invar(g)) /\ reachable2(allEvts,g) /\
    trans(g,e,g') /\ e=evt_inc(k) => invar(g') %(InvarInc)%
  . (reachable2(allEvts,g) => invar(g)) /\ reachable2(allEvts,g) /\
    trans(g,e,g') /\ e=evt_inc(k) /\ ctrl(g)=s1 /\ cnt(g)+k<4 =>
    invar(g') %(InvarInc)%
```

```
  . (reachable2(allEvts,g) => invar(g)) /\ reachable2(allEvts,g) /\
    trans(g,e,g') /\ e=evt_inc(k) /\ ctrl(g)=s1 /\ cnt(g)+k=4 =>
    ctrl(g')=s2 %(InvarInc)%
  . (reachable2(allEvts,g) => invar(g)) /\ reachable2(allEvts,g) /\
    trans(g,e,g') /\ e=evt_inc(k) /\ ctrl(g)=s1 /\ cnt(g)+k=4 =>
    cnt(g')=4 %(InvarInc)%
  . (reachable2(allEvts,g) => invar(g)) /\ reachable2(allEvts,g) /\
    trans(g,e,g') /\ e=evt_inc(k) /\ ctrl(g)=s2 /\ cnt(g)+k=4 => false
    %(InvarInc)%
  . (reachable2(allEvts,g) => invar(g)) /\ reachable2(allEvts,g) /\
    trans(g,e,g') => invar(g') %(InvarStep)%
  . invar(g) => cnt(g)<=4 %(InvarImpliesSafe)%

  %% the safety theorem for our counter
  forall g:Conf
  . reachable2(allEvts,g) => cnt(g)<=4 %(Safe)%

 axiom false %(thm_false)% %% proof fails as it should


  forall m,n : Nat; g : Conf
  . m = n <=> __==__(m,n) %(thm_eq)%
  . init(g) => ctrl(g) = s1 %(thm_init_ctrl)%
  . init(g) => cnt(g) = 0 %(thm_init_cnt)%
  . 0*m+n = n %(thm_0at0)%
  . 0*9+n = n %(thm_0at0)%
  . 0@@n = n %(thm_0at0)%
end
```

Appendix D

A Heterogeneous Specification for the ATM example

```
logic UMLStateO
spec ATM =
  name ATM;

  vars cardId
     , pin
     , trialsNum
     ;

  inputs userCom.card(c)
       , userCom.PIN(p)
       , bankCom.verified
       , bankCom.reenterPIN
       ;

  outputs bankCom.verify(card, pin)
        , userCom.ejectCard
        , userCom.keepCard
        ;

  states Idle, CardEntered, PINEntered, Verifying, Verified;

  init Idle : [ true ];

  trans Idle --> CardEntered : userCom.card(c) / { cardId:=c };
  trans CardEntered --> PINEntered : userCom.PIN(p) / { pin:=p };
  trans PINEntered --> Verifying :  / { bankCom.verify(cardId,pin) };
  trans Verifying --> Verified : bankCom.verified;
```

```
  trans Verified --> Idle : / { userCom.ejectCard; trialsNum := 0 };

  trans Verifying --> Idle : bankCom.reenterPIN [ trialsNum >= 3 ] / {
    userCom.keepCard; trialsNum := 0 };
  trans Verifying --> CardEntered : bankCom.reenterPIN [ trialsNum < 3
    ] / { trialsNum := trialsNum+1 }
end

spec Bank =
  name Bank;

  vars wasVerified;

  inputs atmCom.verify;

  outputs atmCom.verified
        , atmCom.reenterPIN
        ;

  states Idle, Verifying, VeriSuccess, VeriFail;

  init Idle : [ true ];

  trans Idle --> Verifying : atmCom.verify / { wasVerified := 0 };
  trans Verifying --> VeriSuccess : / { wasVerified :=1 };
  trans Verifying --> VeriFail : ;
  trans VeriSuccess --> Idle : / { atmCom.verified };
  trans VeriFail --> Idle : / { atmCom.reenterPIN }
end

logic UMLComp
spec Composite =
  name ATMSystem;

  comp atm : ATM;
  comp bank : Bank;

  export atm.userCom ;
  conn atm.bankCom -- bank.atmCom
end

logic CASL
spec System =
  {Composite with translation UMLComp2CASL}
```

```
  and
  {ATM  with translation UMLStateO2CASL}
  and
  {Bank with translation UMLStateO2CASL}
end
```

Appendix E

Language Definitions for UMLCOMP

E.1 Abstract Syntax

```
BASIC_SPEC  ::=  basic-spec ⟨DECL+ ⟩
   DECL  ::=  name-decl ⟨COMP_NAME⟩
           |  mach-decl ⟨MACHINE_DECL⟩
           |  export-decl ⟨EXPORT_DECL⟩
           |  conn-decl ⟨CONN_DECL⟩
   MACHINE_DECL  ::=  MACHINE_NAME MACHINE_TYPE
   EXPORT_DECL   ::=  MACHINE_NAME PORT_NAME
   CONN_DECL     ::=  MACHINE_NAME PORT_NAME MACHINE_NAME PORT_NAME
```

E.2 Concrete Syntax

```
BASIC_SPEC  ::=  DECL;...;DECL
   DECL  ::=  COMP_NAME
           |  MACHINE_DECL
           |  EXPORT_DECL
           |  CONN_DECL
   MACHINE_DECL  ::=  comp MACHINE_NAME : MACHINE_TYPE
   EXPORT_DECL   ::=  export MACHINE_NAME.PORT_NAME
   CONN_DECL     ::=  conn MACHINE_NAME.PORT_NAME  MACHINE_NAME.PORT_NAME
```

E.3 Static Semantics

$\Sigma = (nameS, machinesS, machineTysS, exportsS, connsS)$

$$\frac{nameS = \{\}}{(nameS, machinesS, machineTysS) \vdash \texttt{name-decl}\ \langle name \rangle \triangleright (\{name\}, \{\}, \{\}, \{\}, \{\}, \{\})}$$

$$\frac{mach \in machinesS}{(nameS, machinesS, machineTysS) \vdash \texttt{mach-decl}\ \langle name\ type\rangle \triangleright (\{\}, \{mach\}, \{type\}, \{\}, \{\}, \{\})}$$

$$\frac{m \in machinesS}{(nameS, machinesS, machineTysS) \vdash \texttt{export-decl}\ \langle m\ p\rangle \triangleright (\{\}, \{\}, \{\}, \{\}, \{m\ p\}, \{\})}$$

$$\frac{\{m1, m2\} \subseteq machinesS}{(nameS, machinesS, machineTysS) \vdash \texttt{conn-decl}\ \langle m1\ p1\ m2\ p2\rangle \triangleright (\{\}, \{\}, \{\}, \{\}, \{\}, \{m1\ p1\ m2\ p2\})}$$