Practical Model Transformation from Secured UML Statechart into Algebraic Petri Net

Qin ZHANG and Vasco SOUSA
Laboratory for Advanced Software Systems
University of Luxembourg
6, rue R. Coudenhove-Kalergi, Luxembourg

TR-LASSY-11-08

1 Introduction

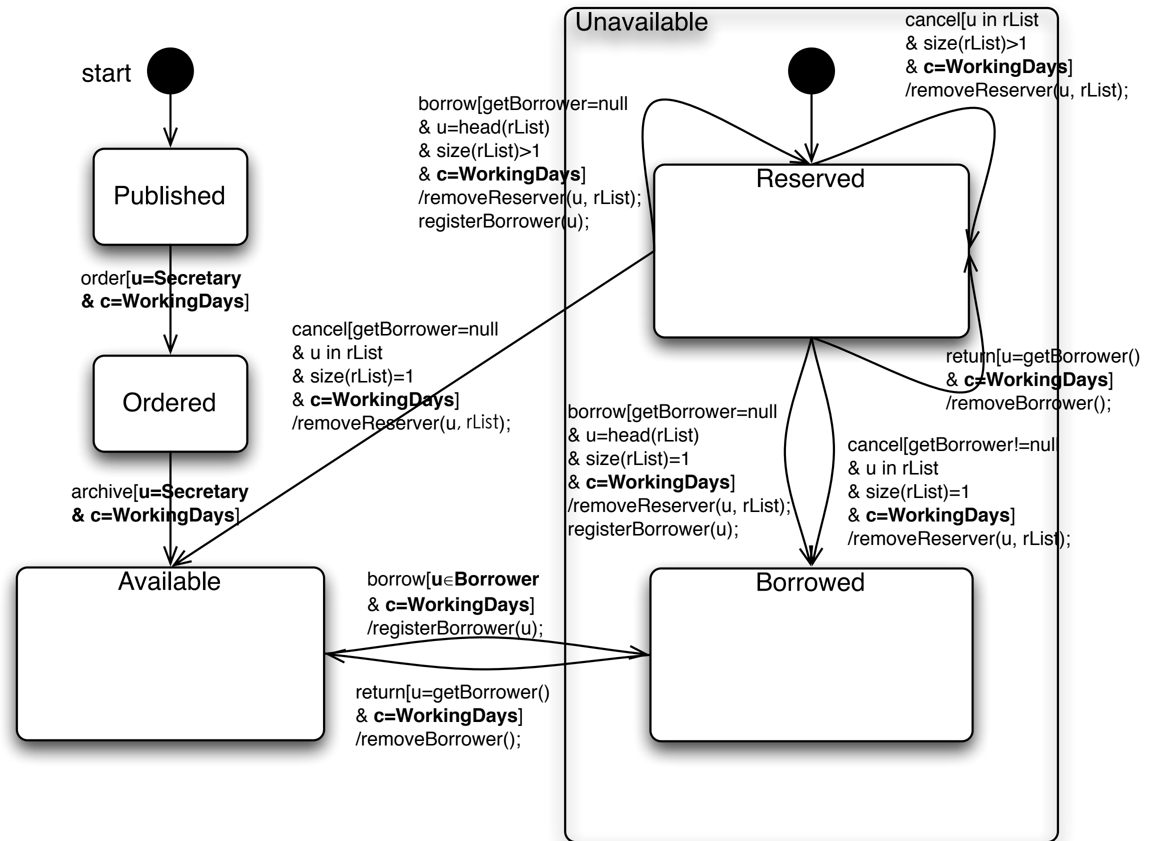
This is a technique report about model transformation from secured UML Statechart into Algebraic Petri Net, a practical approach. We use a simple example, object BOOK in library management system, to illustrate the practical rules of model transformation.

Figure 1 exhibits the secured UML statechart for the logic of the BOOK. In this statechart, the highlighted content is related to access control policies.

2 Model Transformation Rules

In this section, we illustrate practical rules for model transformation, from secured UML statechart into Algebraic Petri Net.

u : User;
rList : List(User); //reservation list
c : Context;
Borrower ⊆ **User;** // Borrower is a subclass of User, consisting of Teachers and Students



Access Control Policies:

Permission(Secretary, Order, Book, WorkingDays)
Permission(Secretary, Archive, Book, WorkingDays)
Permission(Borrower, Borrow, Book, WorkingDays)
Permission(Borrower, Return, Book, WorkingDays)
Permission(Borrower, Reserve, Book, WorkingDays)
Permission(Borrower, Cancel, Book, WorkingDays)

reserve[u!=getBorrower()
& u not in rList
& **c=WorkingDays**]
/registerReserver(u, rList);

***Missing*: u∊Borrower**

Figure 1: UML statechart of object BOOK

2.1 Decompose Composite State

Since the target model, Algebraic Petri Net, of the transformation doesn't have hierarchical structure, we need first decompose the Composite State in the UML statechart, to form a flat structure.

Rule 1: Regarding the hierarchical nested states (Composite State), transit each incoming transition pointed to the composite state to the inner start state; split each outgoing transition starting from the composite state into transitions that each of them starting from one sub state; relocate the Entry action of the composite state to each incoming transitions, either pointing to the composite state or to sub state, as their event actions; relocate Exit action of the composite state to transitions starting from the composite state or from sub states but crossed the boundary of the composite state. Rename the inner start state by adding the original composite state name, which makes it distinguishable with the start state of the whole chart.

2.2 Transform Variables and Related Methods

UML statechart is a sub package of object-oriented technique. It may contain some variables, e.g. parameters of events, and related class methods. They will be transformed into according Abstract Algebraic Data Types (ADTs hereinafter) and other components in APN model.

Rule 2: With regard to each *type* of event parameters and extended state variables in the UML statechart, create a corresponding ADT with generators that may generate sufficient data. Then build an according place holding sufficient initial data (tokens). Finally define an according APN variable for it.

In the Rule 2, the sufficient data may be got from other sub packages of the object-oriented specification, e.g. the class diagram. The usual way is to enumerate possible instances of related classes.

Rule 3: With regard to each class method related with an event parameter or extended state variable (normally get/set method) in the UML statechart, extend the corresponding ADT definition of that variable by adding an operation to implement the logic of the class method, then build a related place with this data type and fill sufficient initial tokens (This place simulates the memory for the variable, so the initial data is usually empty).

In the Rule 3, the logic of added operation in ADT is normally getting or setting a value. Notice that sometimes the logic of a class method is to empty the value in a variable, which can be equally treated as setting a *NULL* value to this variable. So when extending the ADT definition of that variable in APN model, besides adding an operation, you also need to supplement a special generator for the data *NULL*.

```
import "boolean.adt"

Adt user

Sorts user;

Generators
    Null: user; // for removeBorrower() method
    Secretary: user;
    Teacher: user;
    Student: user;

Operations
    isBorrower: user -> bool; // implementing "u \in Borrower"

Axioms
    isBorrower(Null)=false;
    isBorrower(Secretary)=false;
    isBorrower(Teacher)=true;
    isBorrower(Student)=true;
```

Figure 2: Abstract Algebraic Data Type definition of event parameter “user”

Figure 2 shows an example of the ADT definition of the type “user” in the UML statechart of object BOOK.

2.3 Traceability, Concurrency-breaking and Security

There are some features, mainly traceability, concurrency-breaking and security, we need to consider when transforming the secured UML statechart into APN model.

First, under some circumstances, we need to trace a certain component in the UML statechart after we transformed it into APN model, e.g. locating an error in UML statechart after model checking of APN. The most important information we want to trace may be the hazard state and its following transition which lead to an error.

Notice that there may be some events having duplicated names in the UML statechart, see events “borrow”, “return” and “cancel” in Figure 1. These events with duplicated names are actually split from one activity, but lead the object BOOK to different states based on the variant conditions. However, the access control policy is defined on the *activity*, not on the events. So in the future transformed APN model, the security property, transformed from the access control policy on the activity, is the same on the transitions transformed from these events. If the property doesn't hold by APN model during model checking,

```
Adt eventname

Sorts eventname;

Generators
    Start_start_Published: eventname;
    Published_order_Ordered: eventname;
    Ordered_archive_Available: eventname;
    Available_borrow_Borrowed: eventname;
    Borrowed_return_Available: eventname;
    Reserved_cancel_Available: eventname;
    Reserved_reserve_UnavailableStart: eventname;
    Borrowed_reserve_UnavailableStart: eventname;
    UnavailableStart_start_Reserved: eventname;
    Reserved_borrow_Reserved: eventname;
    Reserved_cancel_Reserved: eventname;
    Reserved_return_Reserved: eventname;
    Reserved_borrow_Borrowed: eventname;
    Reserved_cancel_Borrowed: eventname;
```

Figure 3: ADT of distinguishable event transitions in APN model

we need to trace back to which event is unsecured. Therefore we need to make the events with duplicated names distinguishable in transformed APN model.

We noticed one aspect of generic state machine that all states are unique, which means no redundant state names. Further more, the events in the state machine may have duplicated names, but there are no redundant events between the same ORDERED-pair of states (here ORDERED means the first state is the source while the second is the destination). Thus we are able to add source and destination state names to each event, which makes the events distinguishable.

Rule 4: Build a new ADT to rename all events in the UML statechart, which make them distinguishable by a naming format:

$$SourceStateName_EventName_DestinationStateName.$$

Figure 3 gives the example of how to define the ADT of distinguishable event transitions in APN model. Normally we create a generator for each event in the UML statechart, to rename the according transition in the transformed APN model abiding the naming format in Rule 4.

Besides the *traceability*, we also need to consider *concurrency* in future transformed APN model. It is well-known that Petri Net is a mathematical modeling language which is well suited for description of concurrent behavior of distributed systems. Execution of Petri Nets is nondeterministic: when multiple transitions are enabled at the same time, any one of them may fire. If a

transition is enabled, it may fire, but it doesn't have to. However, the UML statechart doesn't hold the concurrency as that in APN model. At any time, there is only an adjacent set of states, pointed by the state transitions from current state, is available to be reached, depending on the nondeterministic event occurrence. Therefore, we need to eliminate the concurrency feature of the transformed APN model, to insure its semantic logic doesn't exceed that of original UML statechart.

To achieve the concurrency-breaking goal, we think about defining a special token, called *indicator*, to restrict which transition may fire so that "indicating" according active state in the original UML statechart. The indicator token is used to insure the simulation of transitions among states in UML statechart: at any time, there is one and only one indicator token in the APN model that makes only the transitions following the place holding this indicator token may fire. This simulation mechanism breaks the concurrency of APN model that insures its semantic logic in accordance with source UML statechart.

Finally, to check the security properties, transformed from access control policies, in future APN model, we need to record corresponding necessary information when a secure transition fires, e.g. user and context. It's naturally to think that the *indicator* is a proper component in the APN model to realize this objective for security checking purpose, since it doesn't conflict with the concurrency-breaking feature when we add some security related information to this special token.

In summary, since the transitions in APN model are distinguishable, according to Rule 4, if one transition fired but led to a hazard state, we could quickly trace back to the unsecured or faulty event in the UML statechart by parsing the transition's name (it contains the information of source state name, event name and destination state name in the UML statechart). Thus we need to record the transition name somewhere when it fires. Meanwhile, for checking the security properties, transformed from the access control policies, we also need to record necessary bounded input of the fired transition, e.g. user and context, as a "log". To make the transformed APN model simple and clear, we adopt these "record" functions into *indicator*, which makes it no longer a black token that only restricting the concurrency of the transformed APN model, but with additional functionality which more likely to be a "runtime status cache".

Rule 5: Define a special ADT called *indicator*, which consists of three fields that are able to record *actor* and *context*, according to the relevant element fields defined in access control policies, as well as the renamed event names according to Rule 4.

Figure 4 shows an example definition of indicator for future transformed APN model.

```
import "user.adt"
import "context.adt"
import "eventname.adt"

Adt indicator

Sorts indicator;

Generators
    I: user, context, eventname -> indicator;

Operations
    getUser: indicator -> user;
    getContext: indicator -> context;
    getEventname: indicator -> eventname;

Axioms
    getUser(I($u, $c, $en))=$u;
    getContext(I($u, $c, $en))=$c;
    getEventname(I($u, $c, $en))=$en;

Variables
    u: user;
    c: context;
    en: eventname;
```

Figure 4: ADT definition of indicator

2.4 Construct Backbone Structure of APN model

After decomposition of the nested states in the original UML statechart into a flat structure, definition of ADTs for data types of event parameters, extended state variables, event names and indicator, we are able to build the backbone structure of the target APN model.

Rule 6: With regard to each state, including "Start" and "End" states, in the UML statechart, build a corresponding place in APN with the same state name. Since these places represent the original states in UML statechart, we name them *state-places*. All these *state-places* are assigned a multiple set of the type *indicator*. Finally initialize one and only one indicator token in the state-place corresponding to the original Start state in the UML statechart.

Rule 7: With regard to each event (state transition) in the UML statechart, build a corresponding APN transition and related input/output arcs from/to relevant *state-places*, which consume/produce an indicator token respectively. The APN transition is named according to the format in Rule 4.

Figure 5 shows the backbone APN model structure transformed from the UML statechart in Figure 1.

2.5 Extract Entry/Exit Actions in the States

In UML statechart, there is a special mechanism of modeling actions that deeply coupled with states, called entry/exit actions. If transformed into APN model, we need to extract these actions out of states.

Rule 8: With regard to entry actions of a state in the UML statechart, build a corresponding transition sequence and insert it between the state-place and its previous transitions with necessary intermediate places supplemented. These supplemented places consist of a start place indicating the beginning of event actions and several intermediate places between pairs of transition in the sequence. Re-direct all the input-arcs of this state-place to the start place of the new transition sequence. All these supplemented places are assigned a multiple set of type *indicator*.

Rule 9: With regard to exit actions of a state in the UML statechart, build a corresponding transition sequence and insert it between the state-place and its following transitions (re-direct the output-arcs of this state-place to the start place of the new transition sequence). The creation of the transition sequence is the same as the way in Rule 8.

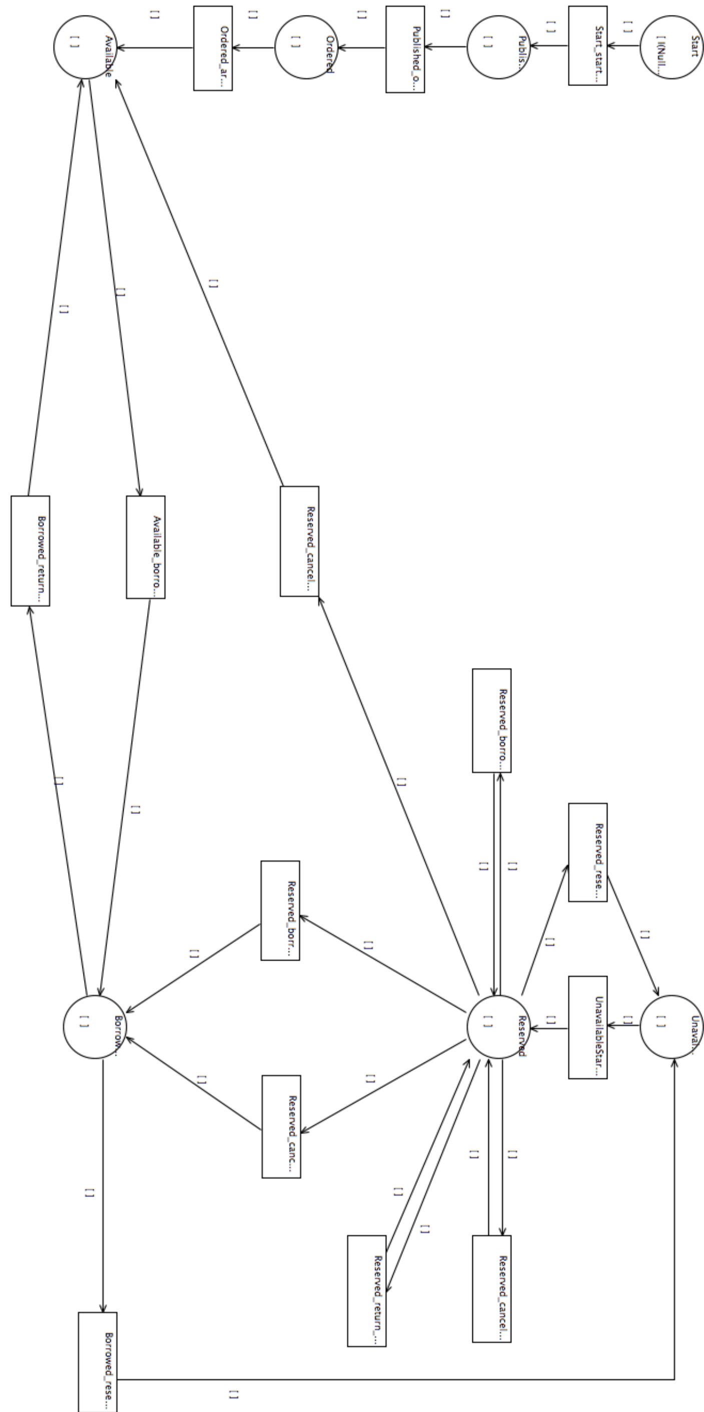


Figure 5: Backbone structure of APN model

2.6 Transform Choice Pseudo State

In the UML statechart, there is a kind of special pseudo states, called choice pseudo state. As this kind of states are used to reason conditions related to the extended state variables, it's more suitable for us to transform them into transitions, not states, with relevant guards.

Rule 10: With regard to each choice pseudo state, build one transition with relevant guard for each of its choice conditions respectively.

2.7 Supplement Transitions and Arcs

At last, we need to supplement the guard of each transitions built already and assign proper argument on each arc.

Rule 11: Supplement each state transition: besides consuming/producing indicator token in the corresponding APN transition, complete it with required data from related place and the guard conditions. There are several important sub rules: (a) Regarding event parameters and extended state variables, in APN model when consumes a corresponding token from the mapped place, produces the same token and put it back to insure the date completeness in that place. The only difference is that if the extended state variable changes in transition firing, make the change when producing the token back. (b) In guard conditions, if there are some special operations, like "$\in$", implement them in ADT by building proper operations and axioms.

Rule 12: With regard to the output-arcs of secure transitions transformed from the events in UML statechart, change the arguments on these output-arcs as the generator of indicator, to produce new indicator token while recording the information of "actor", "context" and transition name.

Rule 13: With regard to actions in an event in the UML statechart, implement them as operations with proper axioms in related ADTs of their parameter types, and then replace the arguments of the output-arcs of the transition, transformed from this event, by the operations to produce appropriate tokens to the places that work as memory for the original methods in the UML statechart.

Figure 6 exhibits the final complete construction of the transformed APN model. Full definitions of ADTs can be found in the section.

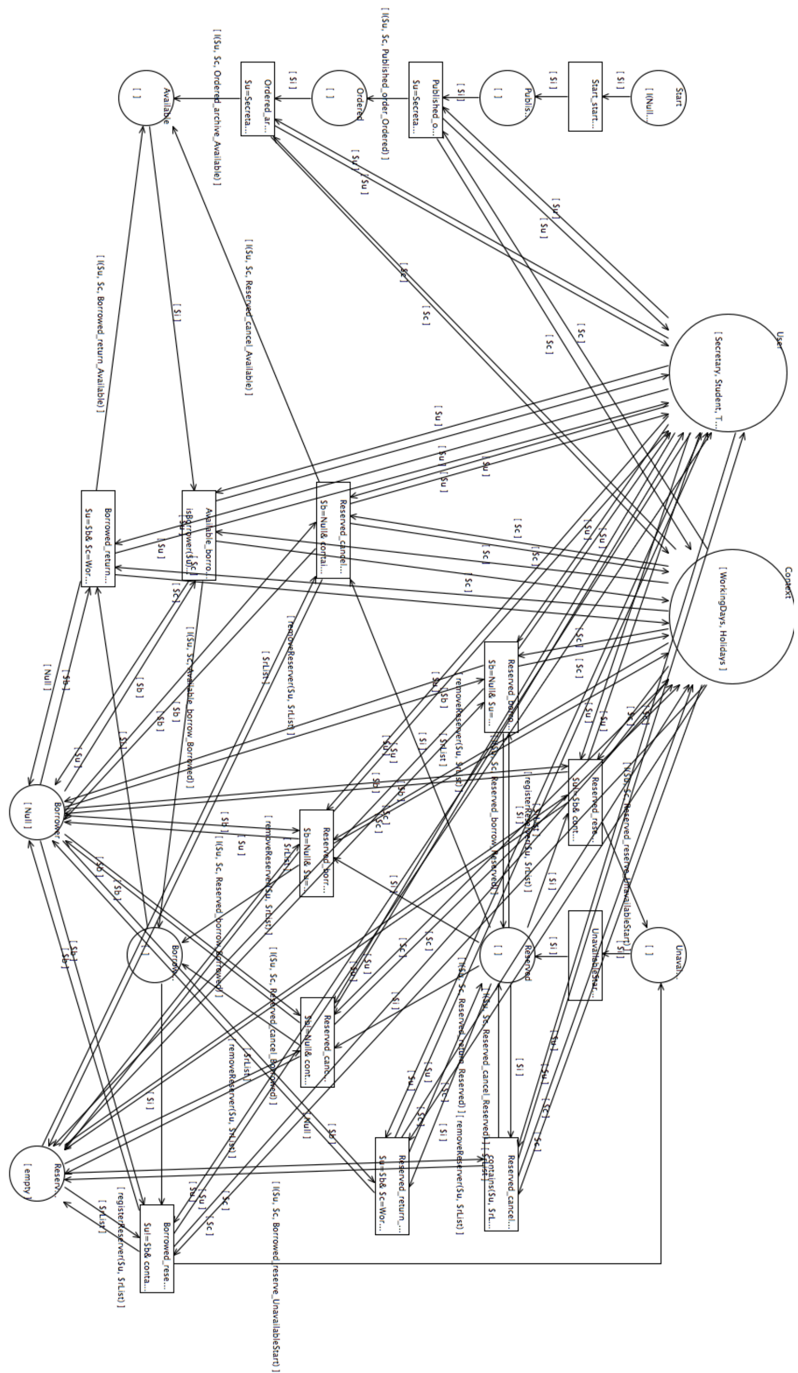


Figure 6: Complete construction of APN model

3 Full Definition of ADTs in transformed APN Model of the Example Library Management System

```
import "boolean.adt"

Adt user

Sorts user;

Generators
    Null: user; // for removeBorrower() method
    Secretary: user;
    Teacher: user;
    Student: user;

Operations
    isBorrower: user -> bool; // implementing "u \in Borrower"

Axioms
    isBorrower(Null)=false;
    isBorrower(Secretary)=false;
    isBorrower(Teacher)=true;
    isBorrower(Student)=true;
```

Figure 7: ADT definition of user

```
import "user.adt"
import "list.gadt"
import "nat.adt"

Adt userlist Is list[user]

Operations
    removeReserver: user, list[user] -> list[user];
    registerReserver: user, list[user] -> list[user];
    headseq: user, list[user], list[user] -> list[user];
    tailseq: user, list[user] -> list[user];

Axioms
    if lt(size($l),suc(suc(suc(zero)))) = true then registerReserver($u, $l) = addEnd($u, $l);
    if ge(size($l),suc(suc(suc(zero)))) = true then registerReserver($u, $l) = $l;
    removeReserver($u, $l) = concat(headseq($u, $l, empty), tailseq($u, $l));

    //head sequence of the list before element $u
    if $u = head($l) then headseq($u, $l, $hseq) = $hseq;
    if $u != head($l) then headseq($u, $l, $hseq) = headseq($u, tail($l), addEnd(head($l), $hseq));

    //tail sequence of the list after element $u
    if $u = head($l) then tailseq($u, $l)=tail($l);
    if $u != head($l) then tailseq($u, $l)=tailseq($u, tail($l));

Variables
    u: user;
    l: list[user];
    hseq: list[user];
```

Figure 8: ADT definition of reservation list

```
Adt context

Sorts context;

Generators
    WorkingDays: context;
    Holidays: context;
```

Figure 9: ADT definition of context

```
Adt eventname

Sorts eventname;

Generators
    Start_start_Published: eventname;
    Published_order_Ordered: eventname;
    Ordered_archive_Available: eventname;
    Available_borrow_Borrowed: eventname;
    Borrowed_return_Available: eventname;
    Reserved_cancel_Available: eventname;
    Reserved_reserve_UnavailableStart: eventname;
    Borrowed_reserve_UnavailableStart: eventname;
    UnavailableStart_start_Reserved: eventname;
    Reserved_borrow_Reserved: eventname;
    Reserved_cancel_Reserved: eventname;
    Reserved_return_Reserved: eventname;
    Reserved_borrow_Borrowed: eventname;
    Reserved_cancel_Borrowed: eventname;
```

Figure 10: ADT definition of renewed event names

```
import "eventname.adt"

Adt activity

Sorts activity;

Generators
    Start: activity; // auto-created for Start and sub Start states
    Order: activity;
    Archive: activity;
    Borrow: activity;
    Return: activity;
    Reserve: activity;
    Cancel: activity;

Operations
    getActivity: eventname -> activity;

Axioms
    getActivity(Start_start_Published) = Start;
    getActivity(Published_order_Ordered) = Order;
    getActivity(Ordered_archive_Available) = Archive;
    getActivity(Available_borrow_Borrowed) = Borrow;
    getActivity(Borrowed_return_Available) = Return;
    getActivity(Reserved_cancel_Available) = Cancel;
    getActivity(Reserved_reserve_UnavailableStart) = Reserve;
    getActivity(Borrowed_reserve_UnavailableStart) = Reserve;
    getActivity(UnavailableStart_start_Reserved) = Start;
    getActivity(Reserved_borrow_Reserved) = Borrow;
    getActivity(Reserved_cancel_Reserved) = Cancel;
    getActivity(Reserved_return_Reserved) = Return;
    getActivity(Reserved_borrow_Borrowed) = Borrow;
    getActivity(Reserved_cancel_Borrowed) = Cancel;
```

Figure 11: ADT definition of activity

```
import "user.adt"
import "context.adt"
import "eventname.adt"

Adt indicator

Sorts indicator;

Generators
    I: user, context, eventname -> indicator;

Operations
    getUser: indicator -> user;
    getContext: indicator -> context;
    getEventname: indicator -> eventname;

Axioms
    getUser(I($u, $c, $en))=$u;
    getContext(I($u, $c, $en))=$c;
    getEventname(I($u, $c, $en))=$en;

Variables
    u: user;
    c: context;
    en: eventname;
```

Figure 12: ADT definition of indicator

4 DSLTrans Implementation

Like the adaptation from pseudo-code to a particular programing language, a conceptual model transformation needs to be adapted to the particular operational semantics of the transformation language that actually implements that transformation. In this section we present an adaptation of the statemachine to Algebraic Petri Net transformation to a DSLTrans transformation.

DSLTrans is a declarative transformation language, where it is possible to control the operationally of the transformation, by grouping the rules into layers, where the rule application within one layer is non-deterministic, and layers are processed in sequence.

For this, we present here, the source and target metamodels folowed by the produced rules, organized in the order of their execution layers.

4.1 Source Metamodel

As a source metamodel, we use the specification presented in Figure 13. With it, we define the type of Hierarchical Statecharts taken as input of our transformation.

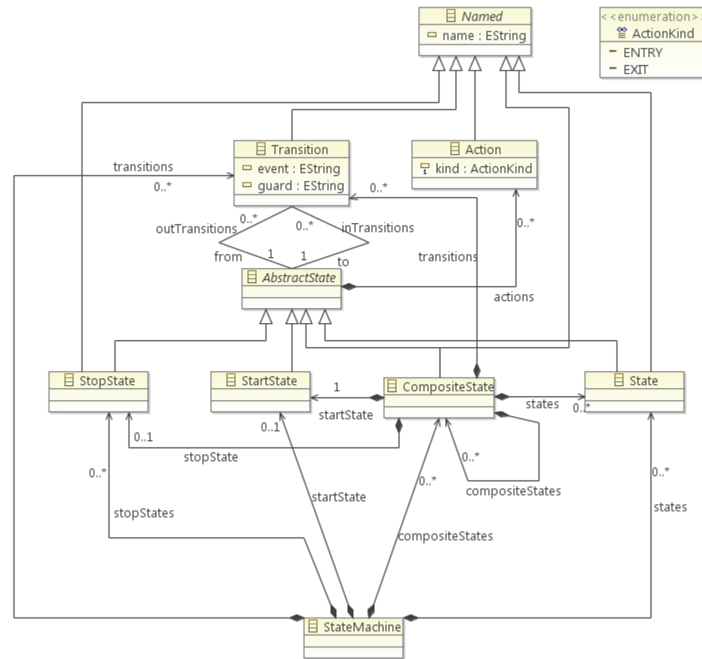


Figure 13: Statechart metamodel

4.2 Target Meta-model

The target of our transformation is the Algebraic Petri Net models accepted by the ALPINA model checker. In Figures 14, 15, 16 and 17, we present the meta models used to define these models. Noticeably in Figure 14, we can observe the structure of the Petri Nets, and in Figure 17, the structure of the Algebraic Data Types.

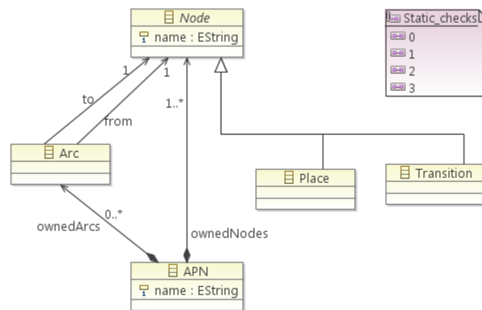


Figure 14: Algebraic Petri Net metamodel

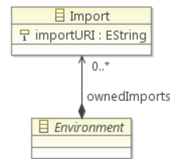


Figure 15: APN Environment metamodel

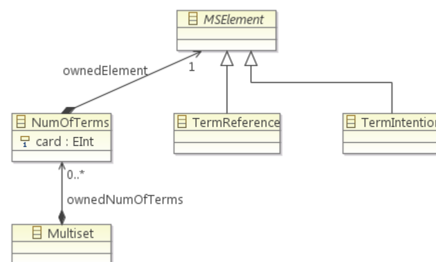


Figure 16: APN Multiset metamodel

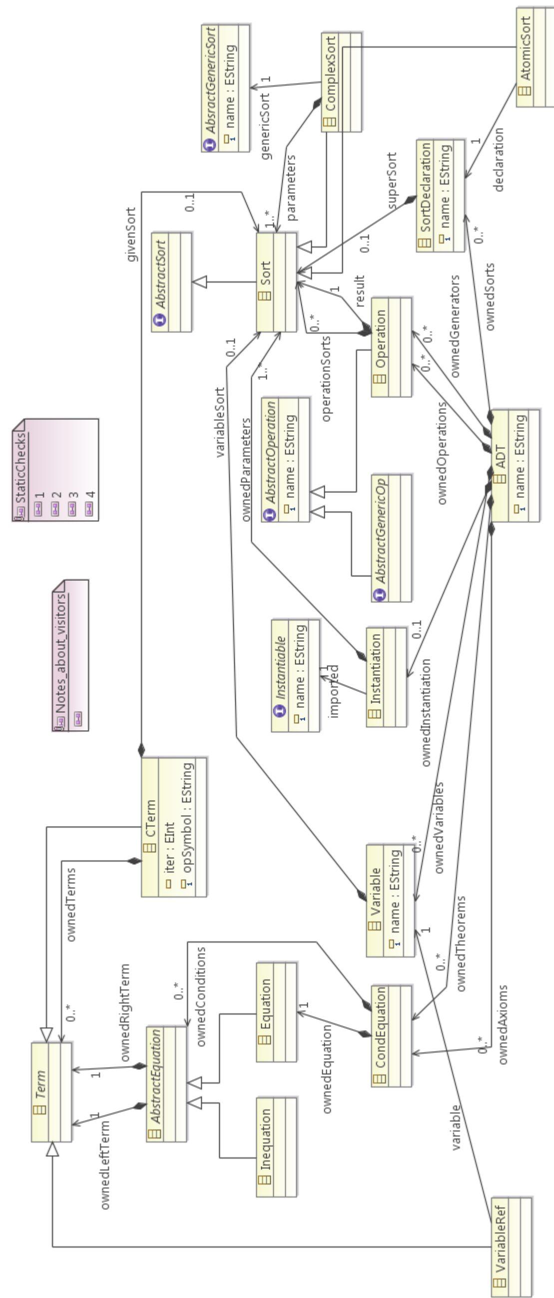


Figure 17: Algebraic Data Types metamodel

4.3 Layer 01

In this layer we have only one rule that ensures, the remaining layers have a node to aggregate the the result of the subsequent layers.

This rule (Figure 18) transforms the root element of a statemachine model into a root of an Algebraic Petri Net model.

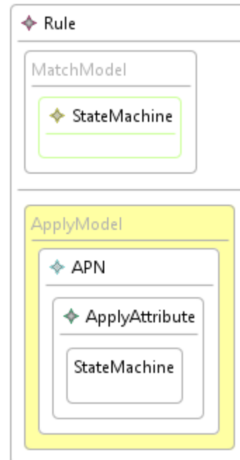


Figure 18: Layer 01

4.4 Layer 02

With this rule we produce the initializer ADT, with the particularity that the result of this layer is serialized into a different file from the one used by the remaining layers of the transformation.

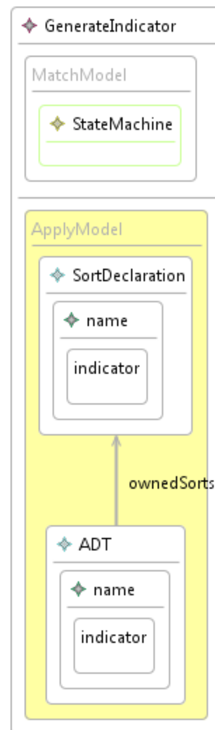


Figure 19: Layer 02

4.5 Layer 03

With this layer we transform the state elements of the statemachine into Algebraic Petri Nets places. In the rules in Figure 20 and the first rule of Figure 21 are direct in their transformation into places. The second rule of Figure 21 does this same transformation, but includes the name of the composite state in the naming of the places resulting from the transformation of a contained start state.

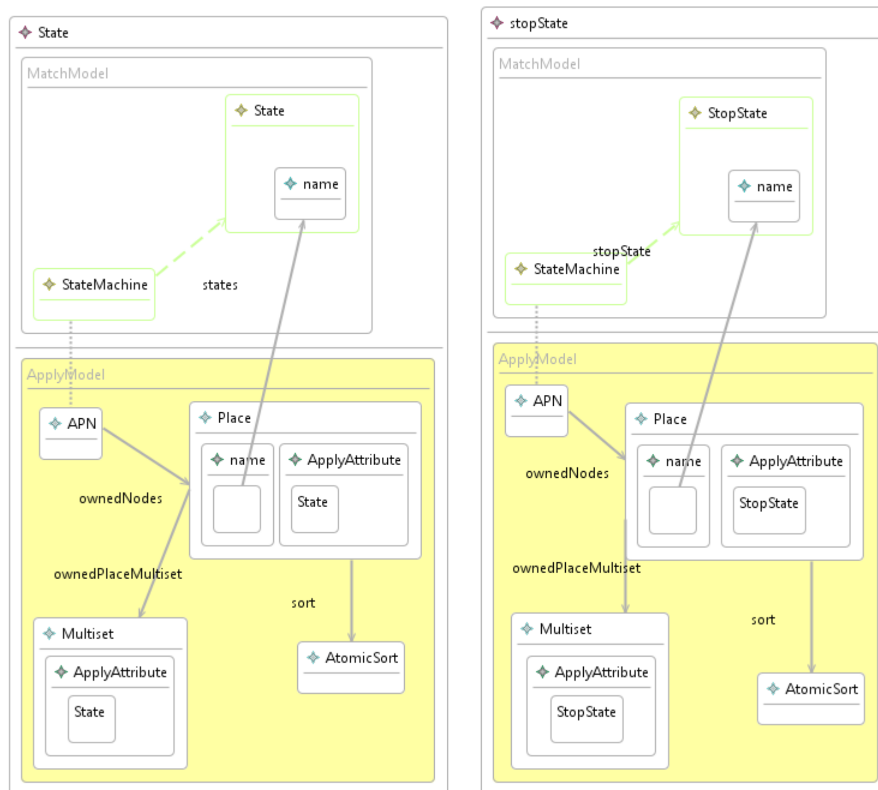


Figure 20: Layer 03

4.6 Layer 04

After producing the places, we transform the transitions linking them to the corresponding places. For this linking process to take place, we take the transition transformation rules to a new layer, allowing us to deal with all the adaptations of the statemachine transitions, to Algebraic Petri Net transitions.

This includes the special cases when a statemachine transition starts from a composite state (Figures 23, 26); when the transition ends in a composite state

(Figure 23, 24 and 25). The remaining rules deal with the simpler cases, where rules in Figures 25, and 27, take care of the particular situation that we cannot derive the naming directly from start states.

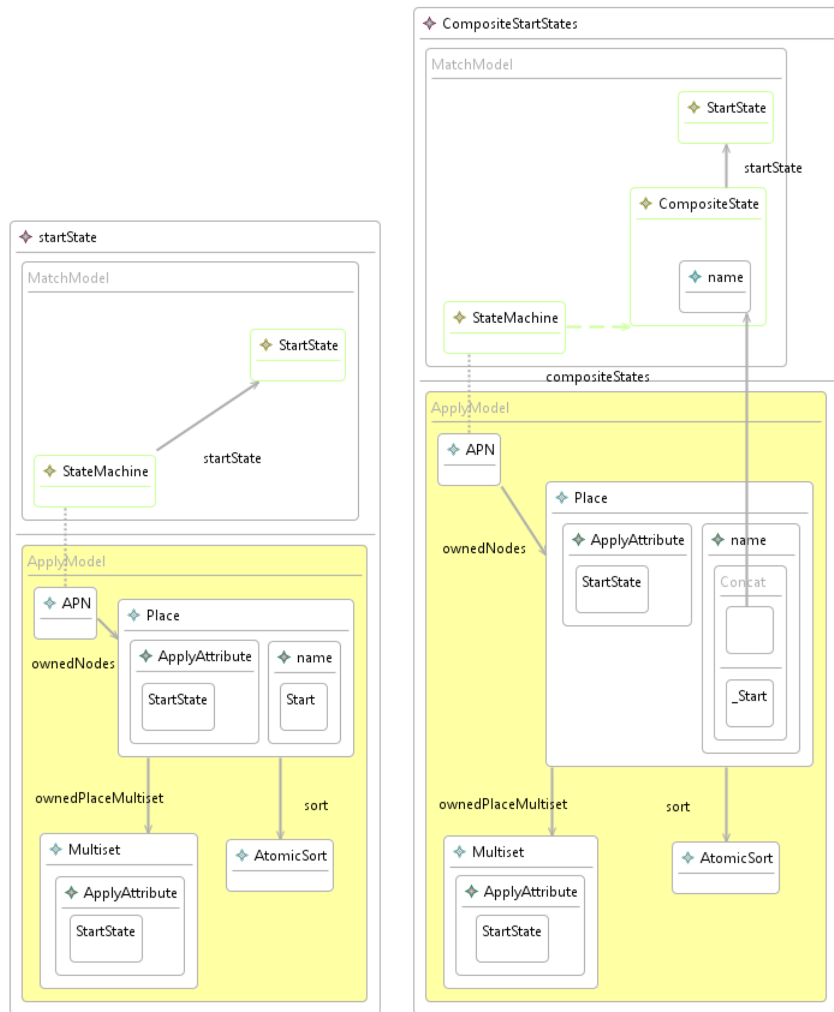


Figure 21: Layer 03

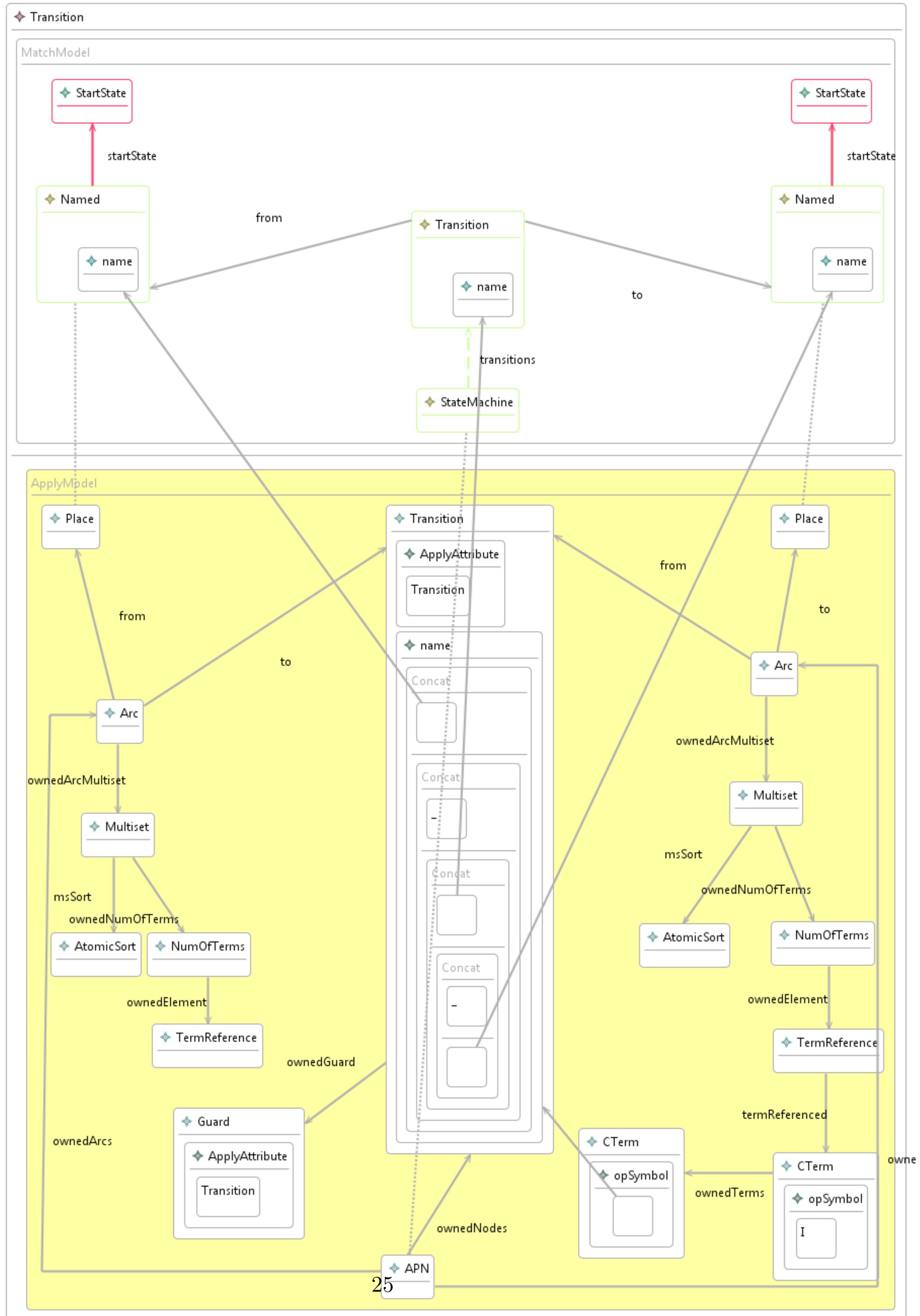


Figure 22: Layer 04

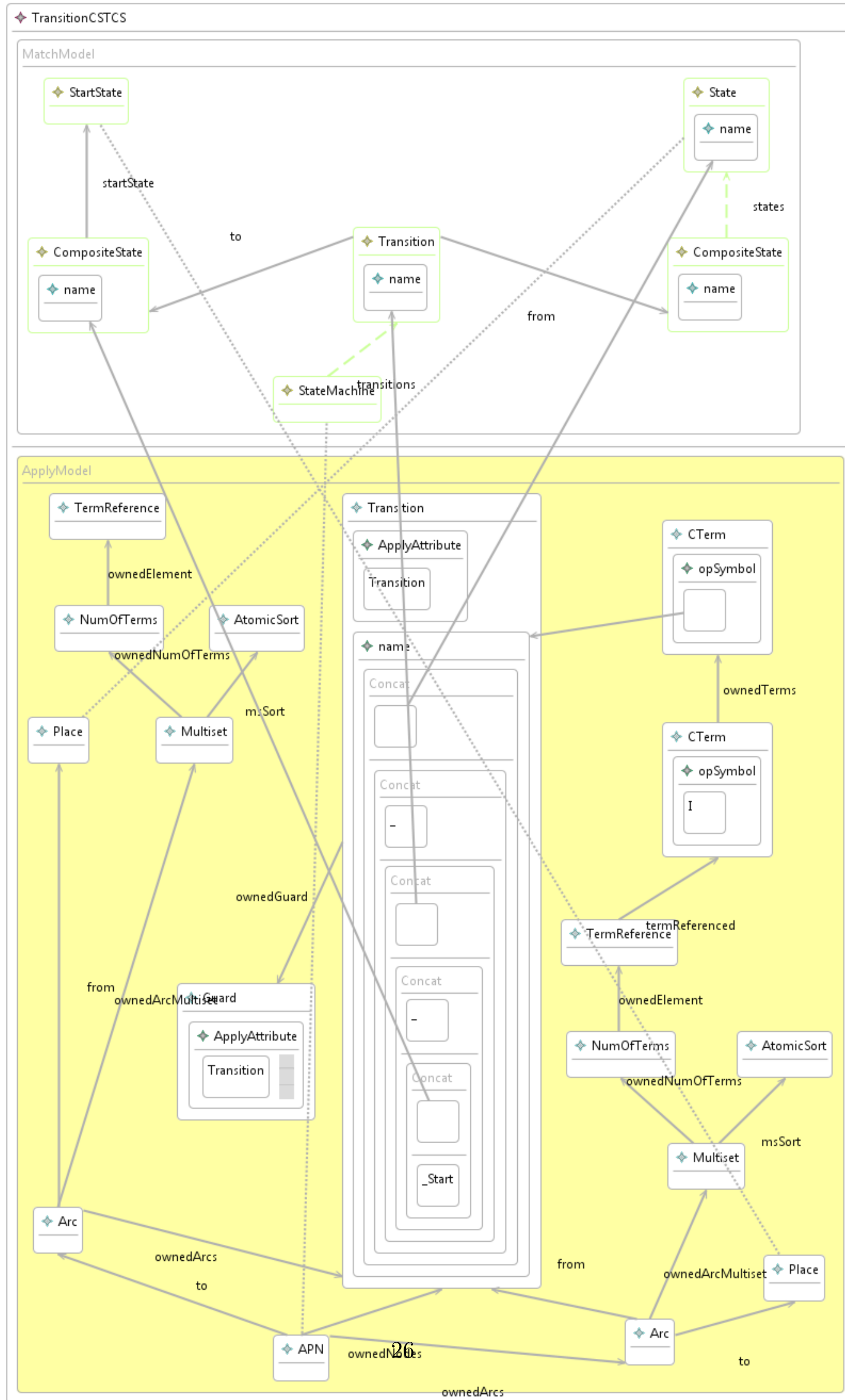


Figure 23: Layer 04

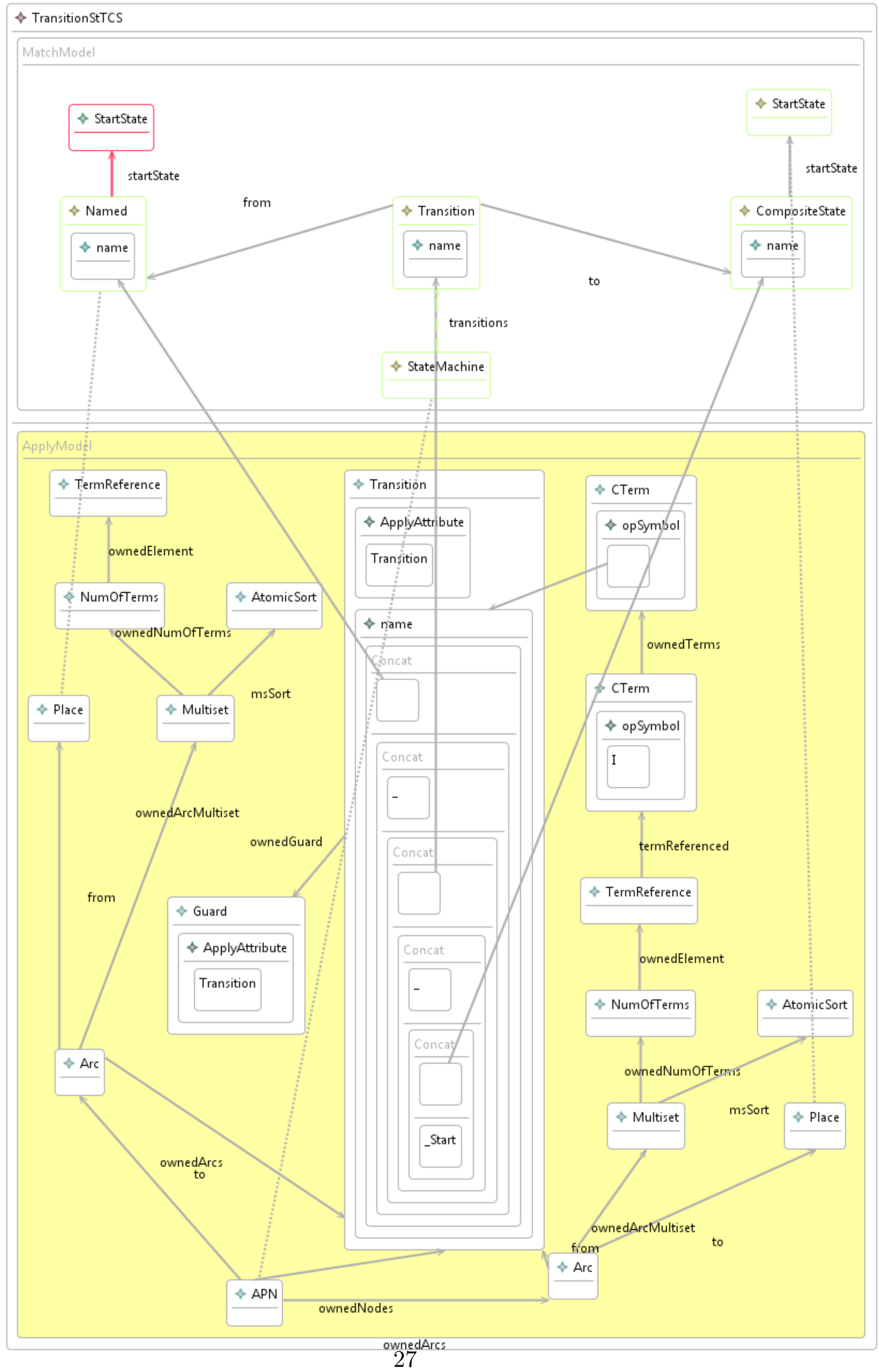


Figure 24: Layer 04

TransitionSTCS

MatchModel

StartState
startState
from
StartState
Transition
name
CompositeState
name
to
transitions
StateMachine

ApplyModel

TermReference
ownedElement
NumOfTerms
AtomicSort
ownedNumOfTerms
msSort
Place
Multiset
ownedArcMultiset
ownedGuard
from
Guard
ApplyAttribute
Transition
Arc
ownedArcs
to
Transition
ApplyAttribute
Transition
name
Concat
Start_
Concat
Concat
_
Concat
_Start
APN
ownedNodes
ownedArcs
CTerm
opSymbol
ownedTerms
CTerm
opSymbol
I
termReferenced
TermReference
ownedElement
NumOfTerms
AtomicSort
ownedNumOfTerms
Multiset
msSort
Place
from
ownedArcMultiset
Arc
to

Figure 25: Layer 04

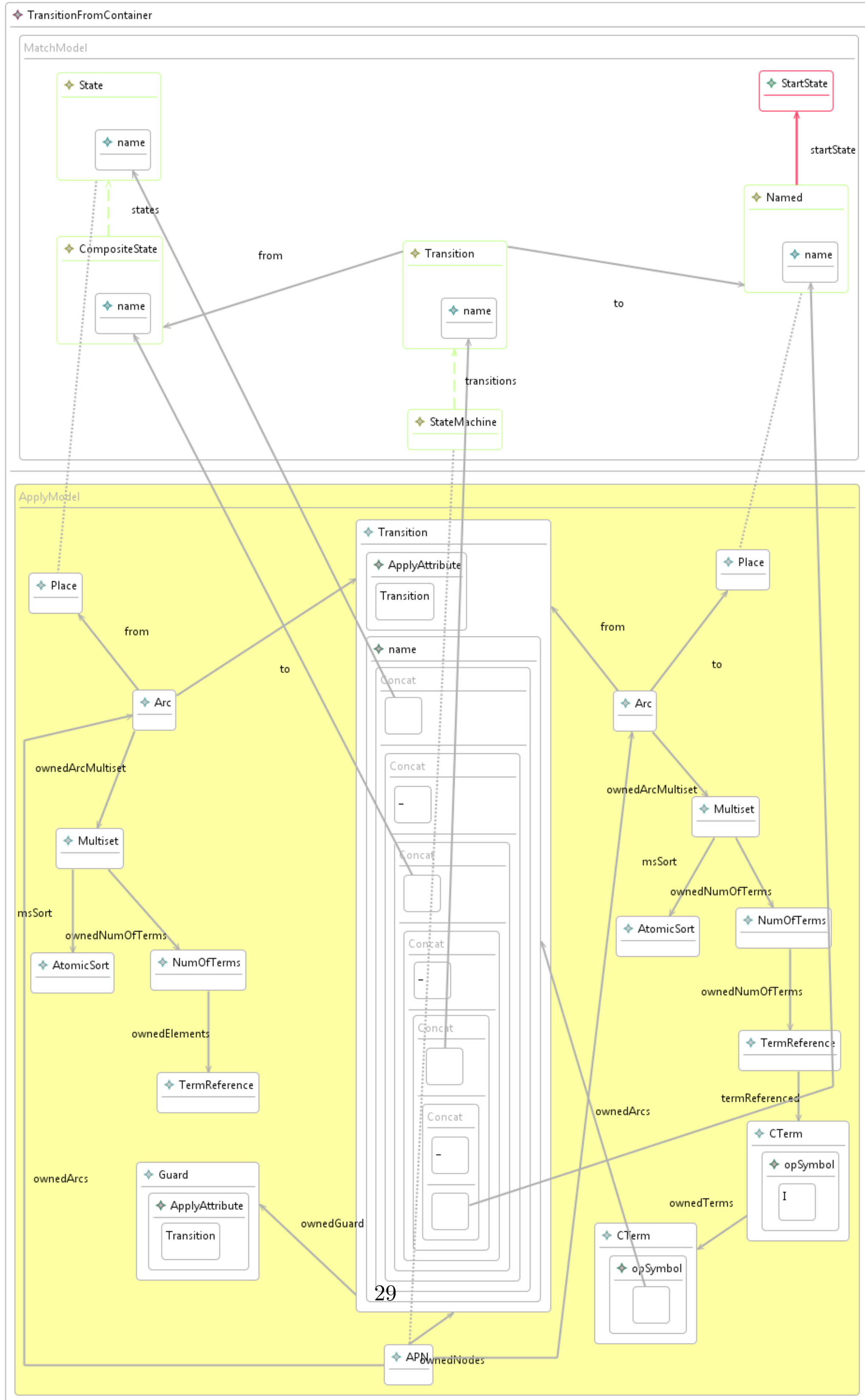


Figure 26: Layer 04

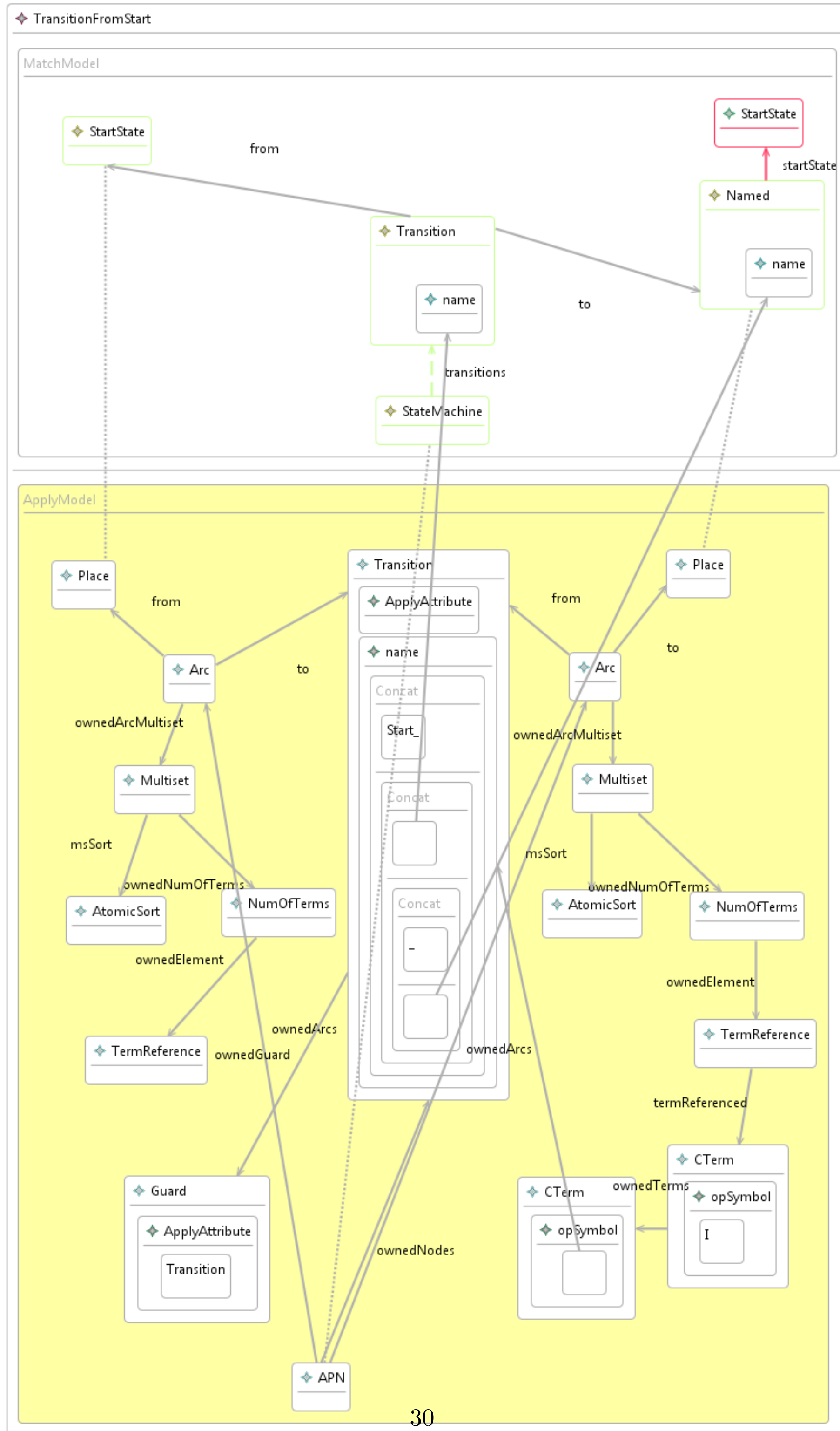


Figure 27: Layer 04

4.7 Layer 05

In this layer we deal with the transition actions in the first rule of Figure 28. The rules in Figure 29 deal with the node actions and transits its transformation to the preceding transition or subsequent transition depending on whether the actions are entry or exit actions. In the rules of Figure 30 we do the same, but this time for when these actions are to be applied in a indirect form through the composite states. Finally the second rule in Figure 28 transcribes the transition guards, from the statecharts to the Algebraic Petri Net.

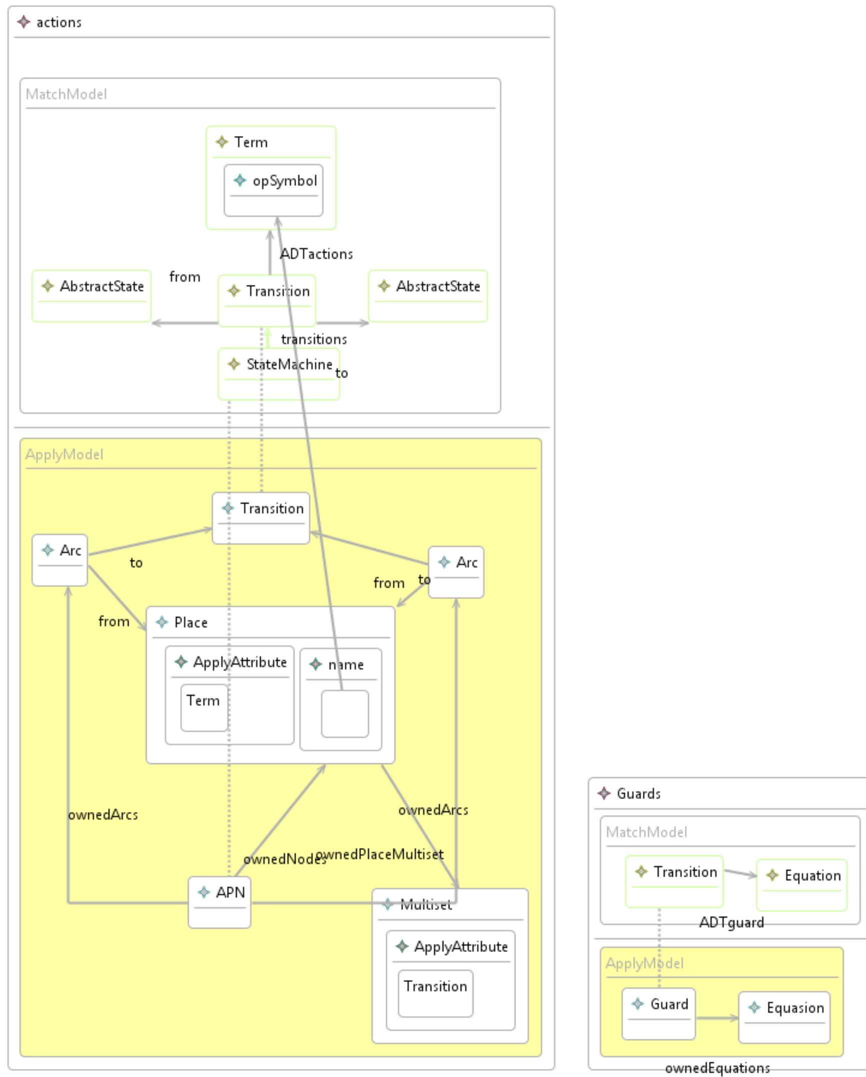


Figure 28: Layer 05

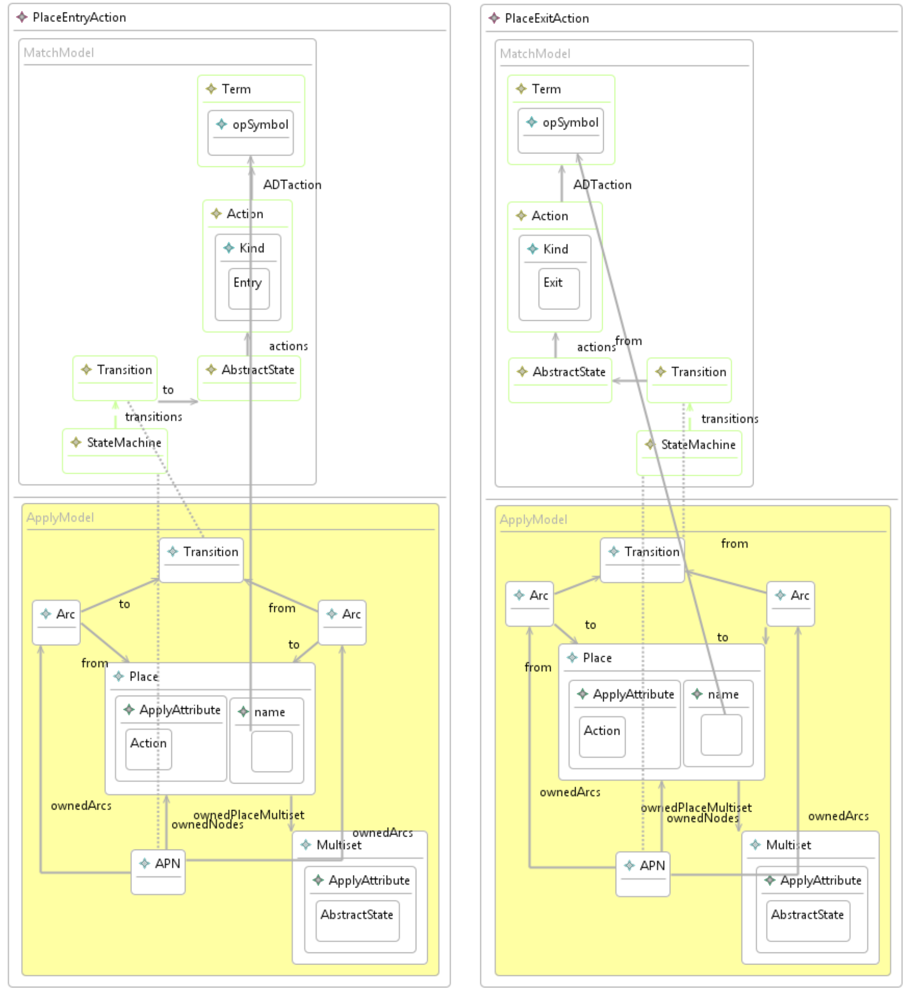


Figure 29: Layer 05

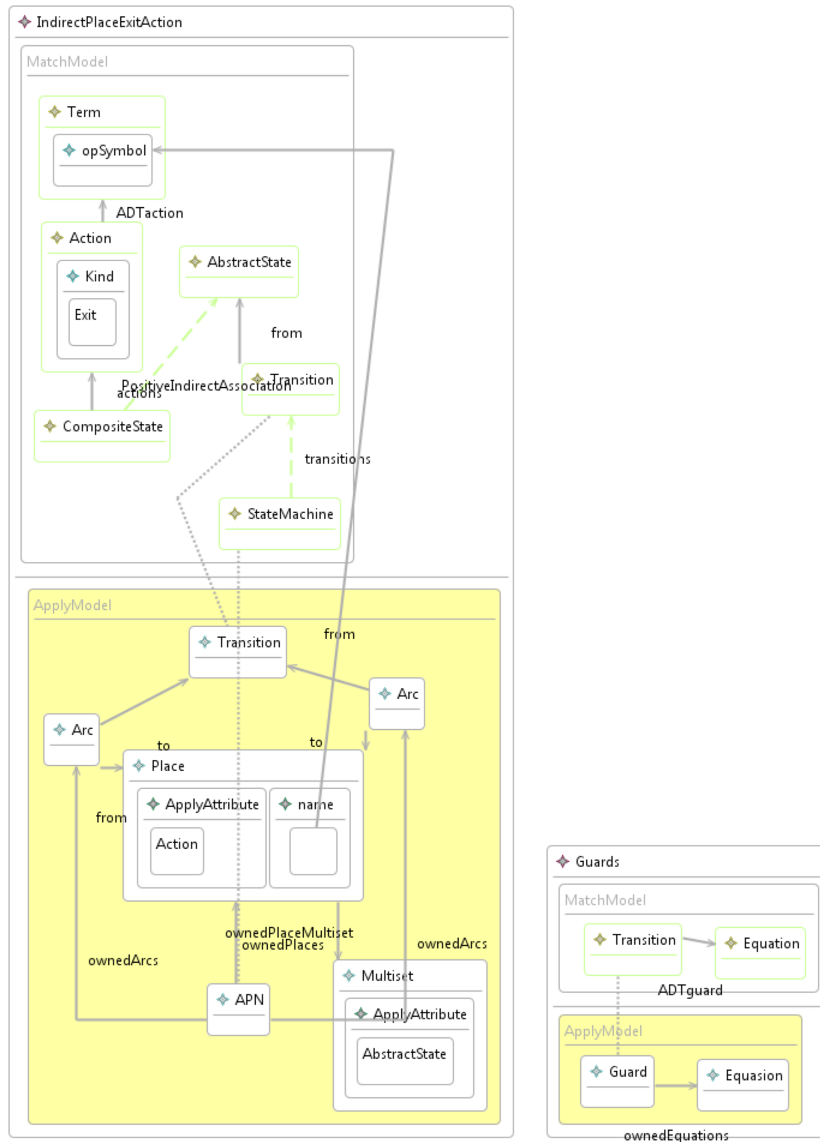


Figure 30: Layer 05

4.8 Layer 06

In this final layer we set the imports present in state and transitions in the transformed model.

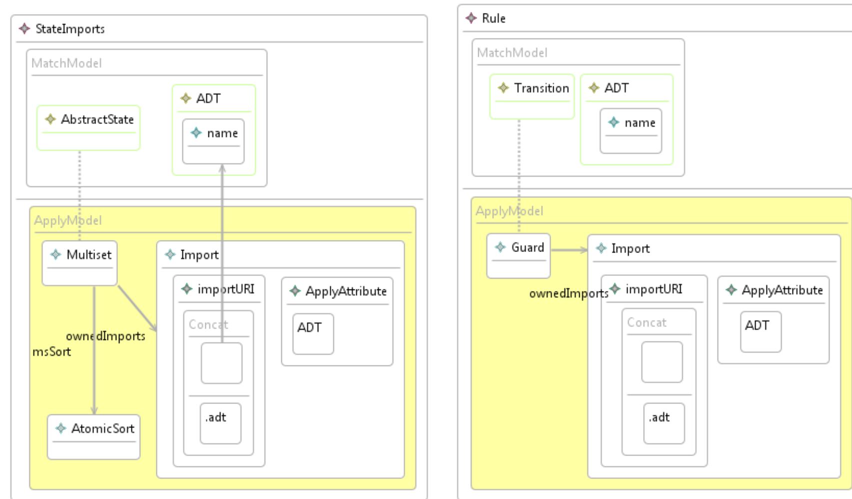


Figure 31: Layer 06