Concurrent Transaction Frame Logic Formal Semantics
for UML Activity and Class Diagrams

Franklin Ramalho\textsuperscript{1,2}, Jacques Robin\textsuperscript{2}, Ulrich Schiel\textsuperscript{1}

\textsuperscript{1}Departamento de Sistemas e Computação – Universidade Federal de Campina Grande (UFCG)
Caixa Postal 10106 – 58109-970 – Campina Grande, PB – Brasil

\textsuperscript{2}Centro de Informática – Universidade Federal de Pernambuco (UFPE)
Caixa Postal 7851– 50732-970 Recife, PE - Brasil

franklin@dsc.ufcg.edu.br, jr@cin.ufpe.br ulrich@dsc.ufcg.edu.br

\textbf{Abstract.} We propose Concurrent Transaction Frame Logic (CTFL), an extension of first-order Horn logic that gives declarative semantics to object-oriented deductive databases, as a language to provide formal semantics to UML activity and class diagrams. CTFL extends Horn logic with object-oriented class hierarchy and object definition terms, and with three new logical connectives - serial conjunction, concurrent conjunction and atomic modality - that declaratively capture temporal and concurrency constraints on updates and transactions. CTFL has coinciding, sound and refutation complete proof and model theories. Contrary to language sets previously proposed to provide formal semantics to UML activity diagrams, CTFL allows using a single language to (1) formally and concisely describe the semantics of both activity and class diagrams, (2) automatically verify UML models based on these two diagrams using theorem proving and logic-based model checking and (3) implements the model as an executable, object-oriented logic program.

\section{1 Introduction}

The Unified Modeling Language (UML) \cite{UML} provides an intuitive, visually clarifying standard notation for specifying and modeling computational systems. UML specifications and models are far more precise and less ambiguous than their natural language counterparts. They go a long way into facilitating communication between all the actors involved in the development of a system. However, the current UML standard is merely semi-formal, since its semantics is only defined in natural language rather than in some rigorous mathematical notation. This severely hinders the construction and use of automatic development tools for model verification, behavioral code generation and code testing in UML-based system engineering processes. To overcome this limitation, various proposals have recently been put forward to provide formal semantics to various UML diagrams \cite{Evans1998,Gogolla1998,DeLoach1999,Borger2000,Rodrigues2000,Eshuis2001,Aredo2002,Kuske2002,Varro2002}. These proposals are very diverse in terms of the formal languages they use to describe UML diagrams and the development task automation functionalities that can be provided by tools relying on these languages.
However, those covering Activity Diagrams (AD) share a common tendency to:

- focus only on activity and statechart diagrams, in isolation, *outside of their structural context* provided by Class Diagrams (CD) and other structural diagrams;
- provide only *operational* semantics, which are often seen as helpful in practice essentially to CASE tool developers, with axiomatic semantics better geared towards application designers and denotational semantics better geared towards language designers [Gupta 99];
- rely on structurally impoverished *imperative or functional* formal languages that do not fit well the structure rich Object-Oriented (OO) paradigm used in most UML-based development processes;
- rely on *low-level*, and often quite arcane formal languages [Schmidt 97] that forces the analyst to get into minute algorithmic details, that ought to be abstracted until implementation, or entirely through the use of declarative programming [Schmidt 2001]
- rely on a combination of *several* languages, typically one language to formalize the UML diagram structure, another one to formalize desired temporal properties, another one to implement CASE tools reasoning about models using these two formal notations, and often yet a different one to implement the system under development from its UML model.

As a result, a development team wishing to leverage these proposals to combine the intuitive visual clarity of UML with the rigor, robustness and CASE-tool automation of formal methods, faces a very steep learning curve as well as a significant development time overhead at the modeling stage. Given that time to market is the most critical factor in most real-life development projects, alternative approaches are needed to widen the applicability scope of formal, UML-based development.

In this paper, we propose such an alternative approach to provide formal semantics to UML models. It is based entirely on a variant of First-Order Horn Logic (FOHLL). Although this approach has the potential to provide semantics and CASE tools for the whole of UML, in this paper, we present a preliminary proposal focused on the formal semantics of an AD contextualized by a CD1. We show how Concurrent Transaction Frame Logic (CTFL) [Kifer 1995] [Bonner and Kifer 96] provides formal semantics for both activity and class diagrams. CTFL is the straightforward integration of two orthogonal yet synergetic extensions of FOHLL: Frame Logic (FL), an object-oriented extension dealing with complex structural modeling with inheritance hierarchies, and Concurrent Transaction Logic (CTL), a non-monotonic extension dealing with complex behavioral modeling with concurrent logical database updates, transactions, process communications and temporal execution constraints. Coinciding sound and refutation-complete proof and model theory of FL and CTL are given in [Kifer et al. 1995] [Bonner and Kifer 95] [Bonner and Kifer 1996]. Our approach is based on a mapping between the elements of UML activity and class diagrams and the constructors of CTFL. Through this

---

1 We do not cover here the whole complexity of CD, leaving this topic for a separate publication. Instead, we concentrate on the main features of CD that are relevant to provide context to AD.
mapping, these UML diagrams are given proof theoretical and model theoretical formal semantics in FOHL: that of the CTFL program onto which they are mapped.

The rest of the paper is organized as follows. In section 2, we review the main UML AD and AC elements, illustrating each of them on a simple example model. In section 3, we review the OO and non-monotonic constructs of CTFL illustrating them on the same example. In section 4, we provide a systematic mapping between the elements presented in section 2 and the constructs presented in section 3. This mapping defines our UML activity and class diagram formal semantics proposal. In section 5, we point out the main differences and advantages of our approach as compared to related work. In section 6 we review the contributions of the paper and outline directions for future work.

2 UML Class and Activity Diagrams

UML is a diagrammatic and textual language for specification and modeling in Object-Oriented Software Engineering (OOSE). In OOSE, the key software structure is the class. A class is an encapsulated, generic description of objects with similar structure, behavior, and relationship. An illustrative CD example is given in Fig. 1. It is an extension of the Royal & Loyal (R&L) company information system CD presented in [Warmer and Kleppe 1999]. R&L manages fidelity programs for various companies, offering regular customers diverse bonuses such as air miles or discount points.

A CD specifies the signature of each class, *i.e.*, the attributes used to represent the state of the objects of the class together with constraints on their type, and the methods used to represent the behavior of these objects together with constraints on the type of their input parameters and return value. For example in Fig. 1, the class `customer` models a fidelity program customer with attributes `name` and `title` of type `string`, a boolean attribute `isMale`, a `dateOfBirth` attribute of type `date`, and a `returning method `age()`.

A UML CD also specifies the relationships between the defined classes. There are three main types of relationships: the specialization relationship to specify the hierarchy along which classes inherits attributes and methods, the aggregation and composition relationships to assemble complex objects from simples ones viewed as parts, and the general purpose association for other relationships. These relationships can be labeled with cardinality constraints on the number of elements involved at each end of them. For example in Fig.1, `Earning` and `Burning` transactions are defined as subclasses of the general `Transaction` class, and each member of this class is associated to any number of members of the `CustomerCard` class.

What an UML CD does not specify is the behavior encapsulated in the classes' methods. UML provides various diagrams to this effect. A State Diagram (SD) is essentially a graph that represents of a state machine. Its use is recommended to specify the changes that occur in the attribute values of a single object as a result of invoking its methods and that of other objects. It specifies the conditions that trigger such change and the resulting, new values. An AD is essentially a flowchart. Its use is recommended to represent the state changes that occur in the attribute values of several objects that are involved in the implementation of a use-case [Rumbaugh et al. 1999]. Use-cases are requirement diagrams that divide the functionalities of a system into a set of distinct elementary usages. They describe the actors and purpose involved in each such usage. In strictly OO development, each use case must in the end be implemented by
one method of some class. Hence, AD can also be used to describe the decomposition and
control flow of complex methods implemented by way of invoking methods of objects from
various other classes [Perdita and Poley 2000]. Although all UML diagrams are useful and
complementary for complex system development, use-case, class and activity diagrams can be
views as the minimal core of UML with which simple OO systems can be specified and modeled.
This is why we chose CD and AD as the initial focus of our research on UML model formal
semantics.

![UML class diagram](image)

**Figure 1 – An example of UML class diagram**

An illustrative AD example is given in Fig. 2. It specifies the control flow of the burn
method of the **LoyaltyAccount** class.
An illustrative AD example is given in Fig. 2. It models the realization of the burn method of the LoyaltyAccount class from the CD of Fig. 1. This method itself realizes the use-case of the same name in the R&L system requirement document. An AD is a graph where nodes are activities or control constructs and arcs represents transitions between them. Activities are decomposed into atomic action states than can neither decomposed nor interrupted, and activity states than can be interrupted and further decomposed into sub-actions. Such decomposition can then be represented by a finer-grained AD. A complex activity can thus be modeled by a hierarchy of AD, linked to one another through activity states. Actions states can contain either only the name of the action or both its name a specification of the operation that it executes. Such specification can be precisely written using the Object Constraint Language (OCL) [Warmer and Kleppe]. In the AD of Fig.2 all the activities are action states. Activity states can also include entry and exit actions to be respectively executed before entering and after leaving the state. The control constructs are if/merge pairs, representing conditional branching to mutually exclusive threads and fork/join pairs, representing unconditional concurrent threads. For example the AD of Fig. 2 models that in a given invocation of the burn method, either action GetDesiredServiceItem or action GetGasDiscount will follow the action IsGasOption, whereas, both GetPointsOfServiceItem and GetAvailablePoints will always concurrently follow from GetDesiredServiceItem. Transitions arcs can be labeled by an event whose occurrence triggers the transition from one state to the next or by an guard which specifies a triggering condition.

An AD can also include object flows that link it to a related CD. An object flow associates an activity state to a class. It can specify the class, which objects are input parameters to the activity, returned output of the activity or which attribute values are altered by the activity. For example in Fig. 2, one object flow models that the GetAvailablePoints action takes an object of the class LoyaltyAccount as input parameter and another one that the UpdateLoyaltyAccount action alters the value of the attribute points of that object. These alteration can also be precisely written in OCL.

3 Concurrent Transaction Frame Logic

CTFL is the integration of two orthogonal extensions of FOHL, the subset of classical first-order logic where all formulas are in implicative normal forms [Russell and Norvig 2002] with only one conclusion in each implication. A FOHL formula (also called a logic program) is thus a universally quantified conjunction of implications, each either:

- a definite clause of the form \( c \leftarrow p_1 \land \ldots \land p_n \), where \( c, p_1, \ldots, p_n \) are positive literals
- a fact of the form \( c \leftarrow \text{true} \), where \( c \) is a positive literal (usually abbreviated as \( c \).)
Figure 2 – An example of UML activity diagram
3.1 Frame Logic

In a nutshell, Frame Logic (FL) extends FOHL with two new classes of object-oriented logical terms: class definition terms and object creation terms. A class definition term specifies the superclass of a class together with its proper attribute filler and method return type constraints, following the syntactic pattern:

\[
\text{class::superclass}[...\text{attr}_i\ \text{typO}_i\ \text{type}_i, \\
\ldots,\ \text{meth}_j(...,\text{param}_{jk},...)\ \text{typO}_j\ \text{type}_j...]
\]

There are four typing operators in FL that instantiate the \text{typO}_k in the above pattern: *=>, *=>>, => and =>>>. The presence or absence of the * prefix distinguishes between inheritable and non-inheritable type constraints, whereas the > and >> suffixes indicate whether the attribute is single valued or set valued. An object definition term creates a new object instance of a class and assigns its proper attribute and method return values, follow the syntactic pattern:

\[
\text{object:class}[...\text{attr}_i\ \text{assignO}_i\ \text{value}_i, \\
\ldots,\ \text{meth}_j(...,\text{param}_{jk},...)\ \text{assignO}_j\ \text{value}_j...]
\]

There are four value assignment operators, that instantiate the \text{assignO}_k in the above pattern: *->, *->>, -> and ->>>. They follow the same prefix and suffix conventions than the typing operators. In FL, methods do not have bodies as in imperative OO languages. A method is executed when its return result variable unifies with a value during theorem proving. The only difference between attributes and methods is thus that a method can take input parameters.

Together, FL class definition and object creation terms are called \textit{F-Molecules}. Logical variables can appear in any position inside these molecules: as object name, class name, attribute name, method name, attribute value, method return value or method input parameter. This freedom provides FL with a high-order syntax that allows for very concise meta-level specifications. However, there exists a simple, tractable mapping from any F-Molecule to a conjunction of literals, which guarantees that semantically, FL remains a first-order logic [Yang and Kifer 2000].

In order to illustrate FL more concretely, we give in Fig. 3 four FL facts that represent four classes form the UML diagram of Fig. 1. These facts define \textit{loyaltyAccount} and \textit{transaction} as top-level classes (in which the :superclass element of the pattern is simply omitted) and \textit{earning} and \textit{burning} as two subclasses of \textit{transaction}. They also define the type signature constraints on the attributes of these classes, such as \textit{points} *= filter integer in the \textit{loyaltyAccount} class, and on their methods input parameters and output return value, such as \textit{getServiceOption(string) *-> \{supermarket;fly;gas\}}. This last constraint illustrates the disjunctive value syntax of FL, used here to codify a UML enumeration type. These FL facts also define associations through attributes which types are constrained to other classes, such as \textit{card} *=> \textit{customerCard} in the definition of the \textit{transaction} class.

A pair of proof theory and model theory of FL is given [Kifer, Lausen, and Wu 1995]. In essence, the proof theory consists of one isaReflexivity axiom, three predicate logic inference rules, resolution, factoring and paramodulation, and nine new inference rules covering the object-oriented semantics: isaTransitivity, isaAcyclicity\textsuperscript{2}, subclassInclusion, typeInheritance,

\textsuperscript{2}The FL isa relation corresponds to the UML generalization association between a class and its superclass.
inputRestriction, outputRestriction, scalability, merging and elimination. In essence, the model consists of a Herbrand model over a F-Molecule universe. In the same paper, the two semantics are proven to be coinciding, sound and refutation-complete.\footnote{For lack of space, we cannot go into further details of these theories published in a paper more than 100 page long.}

```
loyaltyAccount[points *= integer, membership *= membership, transactions *= transaction, 
    earn(integer) *= void, burn(integer) *= void, isEmpty() *= void, 
    getPoints() *= integer, updateLoyaltyAccount() *= void, 
    cancelBurn() *= void, completedBurn(integer) *= void, 
    checkPointsAvailability(integer, integer) *= void, 
    getServiceOption(string) *= [supermarket; fly; gas], 
    getDesiredServiceItem(string) *= serviceItem, getGasDiscount() *= void].

transaction[points *= integer, date *= date, status *= [inProgress; cancelled; completed], 
    card *= customerCard, loyaltyAccount *= loyaltyAccount, service *= service, 
    program() *= loyaltyProgram].

burning: transaction[].
earning: transaction[].
```

**Figure 3: FL representation of four classes from the UML diagram of Figure 1**

### 3.2 Concurrent Transaction Logic

In a nutshell, Sequential Transaction Logic (STL) extends FOHL with one new *transactional* connective: n-ary *serial conjunction* $\odot$. Concurrent Transaction Logic (CTL) further extends STL with two additional ones: n-ary *concurrent conjunction* $\otimes$, and unary *atomic modality* $\Theta$. These three connectives allow representing in a purely declarative and logical way declarative temporal constraints on the execution order of logical proofs. They thus allow providing declarative proof-theoretic and model-theoretic semantics to logic programs and database updates and transactions, as well as to multiagent and interprocess communication protocols.

The semantics of these new connectives is based on the logic programming concept of **execution as proof attempt**. The semantics of a serial conjunction $p \odot q$ is: *first* execute $p$; *then*, if the execution of $p$ succeeded (i.e., if it was proven true), execute $q$. If either of the two executions failed, so does $p \odot q$. If they both succeeded, so does $p \odot q$. The semantics of a concurrent conjunction formula $p \otimes q$ is: concurrently execute $p$ and $q$. If either of the two executions failed, so does $p \otimes q$. If they both succeeded, so does $p \otimes q$. In this context, the semantics of classical conjunction $p \land q$ becomes: execute both $p$ and $q$, either sequentially or concurrently in any order. If both succeed, so does $p \land q$. If either one fails, so does $p \land q$. The truth-value tables of these two new conjunctions is identical to that of the classical conjunction $\land$. The difference between the three lies in their execution order constraints: specified and sequential for $\odot$, concurrent for $\otimes$, and unspecified for $\land$. Thus, while $\land$ and $\otimes$ are commutative, $\odot$ is not. The atomic modality connective $\Theta$, prevents the formula within its scope to be partially executed. If one element of an atomic conjunction scoped by $\Theta$ fails, or if its execution is interrupted by some event, the other elements must be rolled back and all the objects that had been changed must be restored to their states prior to the start of the atomic conjunction execution. For example, if $q$
fails in the formula $\Theta(p \otimes q)$, then all the state changes resulting from the execution of $p$ must be undone.

A key characteristic of TL is its deliberate focus on defining complex actions and transactions out of simpler ones. It does not include any atomic change primitives in itself. To be used in practice, it must be parametrized with a set of such primitives. One example of such primitives useful in the illustrative example of this paper are insertion and deletion of logical facts in a logical formula. Horn TL(\{ins,del\}) provides a fully declarative formal semantics for non-monotonic FOHLL, logic programs and database updates and transactions.

A pair of coinciding, sound and refutation complete proof and model theories of STL are given in [Bonner and Kifer 1995]. Their respective extension to CTL is given in [Bonner and Kifer 1998]. The model theory is based on a multipath structure that represent the possible states that a logical database can pass through when of complex transactions of primitive updates composed by the classical logic and transactional connectives of CTL is applied to it. The proof theory is based one axiom stating database invariance through the application of the empty transaction together with four inference rules for transaction definition application, database query, database primitive updating and atomic transaction execution.

3.3 Integrating Frame Logic with Transaction Logic

Given that FL extends FOHLL by introducing new terms and STL and CTL extend it by introducing new connectives, these two extensions are orthogonal and can be straightforwardly combined respectively yielding STFL and CTFL. To be precise, it is CTFL(\{ins,del\}) that we propose as a formal language for UML AD and CD semantics.

While there is no currently available compiler for CTFL, execution platforms are available for two of its subsets: (1) Flora [Yang and Kifer 2000], that compiles and efficiently executes STFL programs, and (2) CTR that interprets CTL programs. Both these platform are implemented as layers on top of the tabled deductive system XSB [Sagonas 1994], a variant of Prolog that relies on an alternative resolution-based FOHLL theorem proving procedure called SLG. This procedure turns XSB both far more declarative and efficient than Prolog. It implements the well-founded semantics for negation as failure and it caches partial proofs to avoid inefficient redundant computation and left-recursion termination problems of Prolog’s SLD resolution.

4 Mapping UML AD and CD to CTFL

To bring to UML AD and CD the formal semantics of CTFL we propose to map the CD elements to the FL constructor and the AD elements to the CTL constructors. We show this mapping associating a CTFL stretch on right (on below) to each block of the UML notation for activity diagrams on left (on above).

---

1 In implicative normal form.
2 For lack of space, we cannot go into further details of these theories published in a paper more than 100 page long.
3 www.cs.toronto.edu/~bonner/
4.1 Mapping Class Diagrams

(1) Class

\[
\text{class1} \{ \text{attribute}_c1 \Rightarrow \text{type1}, \\
\text{method}_c1(\text{par} : \text{Type}_\text{Par}) \Rightarrow \text{Type1} \}.
\]

**Figure 4 – class mapping**

An UML class signature is mapped directly onto a FL class definition term as shown in Figure 4.

(2) Association mapping

\[
\text{class1} \{ \text{attribute}_c1 \Rightarrow \text{type1}, \\
\text{method}_c1(\text{par} : \text{Type}_\text{Par}) \Rightarrow \text{Type1} \}
\Rightarrow
\text{class2} \{ \text{attribute}_c2 \Rightarrow \text{type2}, \\
\text{method}_c2(\text{par} : \text{Type}_\text{Par}) \Rightarrow \text{Type2} \}
\Rightarrow
\text{class1} \{ \text{attribute}_c1 \Rightarrow \text{type1}, \\
\text{method}_c1(\text{par} : \text{Type}_\text{Par}) \Rightarrow \text{Type1} \}.
\]

**Figure 5 – association mapping**

An UML association is mapped onto FL attributes of the associated classes. For example, in Figure 5, the class1 has a set valued attribute referencing the class2 and vice-versa.

(3) Specialization mapping

\[
\text{class1} \{ \text{attribute}_c1 \Rightarrow \text{type1}, \\
\text{method}_c1(\text{par} : \text{Type}_\text{Par}) \Rightarrow \text{Type1} \}
\Rightarrow
\text{class1} \{ \text{attribute}_c1 \Rightarrow \text{type1}, \\
\text{method}_c1(\text{par} : \text{Type}_\text{Par}) \Rightarrow \text{Type1} \}
\Rightarrow
\text{subclass}_c1 : \text{class1} \{ \text{attribute}_c1 \Rightarrow \text{type2}, \\
\text{method}_c1(\text{par} : \text{Type}_\text{Par}) \Rightarrow \text{Type3} \}.
\]

**Figure 6 – Inheritance mapping**

An UML specialization relationship is mapped onto a FL subclass definition term with the “::” operator. The default inheritability of UML attributes and methods is mapped onto the type constraint operators prefixed by * that captures such semantics, as illustrated in Figure 6.

4.2 Mapping Activity Diagrams

We map each branch in an AD onto a CTL clause which conclusion corresponds to the overall activity modeled by the AD. The nodes and transitions of each branch are then mapped onto the premises of the corresponding CTL clause following the mapping rules below.

(1) Mapping action states

\[
A \leftarrow \text{OCLConstraint}
\]
An action state is mapped onto a simple proposition corresponding to the action state name A. If the action state includes an OCL constraint, it is mapped onto the premise of an additional CTL clause with A as conclusion. Since OCL is essentially a logical language, such mapping is direct.

(2) Mapping activity states

\[ (entry \otimes D \otimes exit) \mid event \]

An activity state is mapped onto the concurrent conjunction of a proposition corresponding to the activity state event and a serial conjunction. The serial conjunction as three conjuncts in the following order: the entry action, the complex activity D and the exit action. The entire mapping process is recursively reapplied on the AD modeling D, yielding the addition of one rule for each branch of this AD to the CTL program.

(3) Mapping fork and join

\[ in \otimes D_1 \mid \ldots \mid D_m \otimes out \]

The concurrent nodes between a fork and a join in an AD are mapped directly onto a CTL concurrent conjunction.

(4) Mapping branching

Branching transitions are mapped onto serial conjunctions following the patterns shown below.

(a)

\[ A \]

\[ \diamond \quad \{True\} \]

\[ B \]

\[ A \otimes B \]

(b)

\[ A \]

\[ \sqcap \quad \{False\} \]

\[ B \]

\[ \neg A \otimes B \]

(c)

\[ A \]

\[ \diamond Guard \]

\[ B \]

\[ A \otimes Guard \otimes B \]
4.3 Mapping Object flow

Object flows that links the behavioral AD to the structural CD in UML, are mapped onto predicates linking the behavioral CTL clauses with the structural FL clauses in CTFL.

(a) \[
\begin{array}{c}
\text{object : Class} \\
\rightarrow \\
A
\end{array}
\]  
\[A(\text{Object})\]

An action state input parameter specification object flows are mapped onto a predicate named after the action state and taking the associated object as sole argument.

(b) \[
\begin{array}{c}
A \\
\rightarrow \\
\text{object : Class}
\end{array}
\]  
\[\Theta (\text{ins(\text{Object:Class})})\]

An object creation object flow is mapped onto a CTFL({ins,del}) primitive database update that inserts the object as a new fact.

(c) \[
\begin{array}{c}
A \\
\rightarrow \\
\text{object : Class [state = value]}
\end{array}
\]  
\[\Theta (\text{del(\text{Object:Class[state]})} \otimes \text{ins(\text{Object:Class[state\rightarrow value]})})\]

An object attribute alteration object flow is mapped onto an atomic serial conjunction of two primitive database updates, one that deletes the old value of the attribute, followed by one that inserts the new value. The atomic modality operator surrounds this conjunction so that no intervening event can occur between the deletion and insertion.

Applying these mapping rules to the AD of Fig. 2 results in the following CTFL program, that gives its formal, logical semantics.

(1) **Rule for Path: A0, A10**
\[
\text{burnActivity} \leftarrow \Theta (\neg \text{checkEnrolled(C1:customer}) \otimes \text{cancel Burning})
\]

(2) **Rule for Path: A0, A1, A10**
\[
\text{burnActivity} \leftarrow \Theta (\text{check Enrolled(C1:customer) \otimes \neg check Card(CC:customer Card)} \otimes \text{cancel Burning})
\]

(3) **Rule for Path: A0, A1, A2, A10**
\[
\text{burnActivity} \leftarrow \Theta (\text{check Enrolled(C1:customer) \otimes check Card(CC:customer Card)} \otimes \neg \text{get Points(LA:loyalty Account) \otimes cancel Burning})
\]

(4) **Rule for Path: A0, A1, A2, A3, A9**
\[
\text{burnActivity} \leftarrow \Theta (\text{check Enrolled(C1:customer) \otimes check Card(CC:customer Card)} \otimes \text{get Points(LA:loyalty Account \otimes is Gas Option(SI:service Item) \otimes get Gas Discount})
\]

(5) **Rule for Path: A0, A1, A2, A3, A4, A5, A6, A7, A10**
\[
\text{burnActivity} \leftarrow \Theta (\text{check Enrolled(C1:customer) \otimes check Card(CC:customer Card)} \otimes \text{get Points(LA:loyalty Account \otimes is Gas Option(SI:service Item) \otimes get Desired Service Item(SI)} \otimes (\text{get Points Of Service Item(SI) \mid get Available Points(LA)}) \otimes \neg \text{check Points Availability(SI,LA) \otimes cancel Burning})
\]
(6) **Rule for Path: A0, A1, A2, A3, A4, A5, A6, A7, A8, A11**

\[
\text{burnActivity} \leftarrow \Theta (\text{checkedEnrolled(C1:customer)} \otimes \text{checkCard(CC:customerCard)} \\
\otimes \text{getPoints(LA:loyaltyAccount)} \otimes \neg \text{isGasOption(SI:serviceItem)} \otimes \\
\otimes \text{getDesiredServiceItem(SI)} \\
\otimes (\text{getPointsOfServiceItem(SI)} \mid \text{getAvailablePoints(LA)}) \\
\otimes \text{checkPointsAvailability(SI,LA)} \\
\otimes \text{updateLoyaltyAccount} \otimes \text{completeBurning})
\]

(7) **Rule defining checkEnrolled action state**

\[
\text{checkEnrolled(C1)} \leftarrow \text{C1:customer}.
\]

(8) **Rule for checkCard predicate**

\[
\text{checkCard(CC)} \leftarrow \text{CC:customerCard} \land \text{CC.valid}
\]

Rules 9-11 defining the action states A2, A3 and A4 follow the same pattern than 7-8 above.

(12) **Rule for updateLoyaltyAccount predicate**

\[
\text{updateLoyaltyAccount(LA,SI)} \leftarrow \Theta (\text{Pre} = \text{LA:loyaltyAccount.points} \otimes \text{del(LA.points)} \\
\otimes \text{ins(LA[points} \rightarrow \text{(Pre} - \text{SI.points)})))
\]

(13) **Rule defining cancelBurning action state**

\[
\text{cancelBurning(SI)} \leftarrow \Theta (\text{ins(} \_\text{burning[points} \rightarrow \text{SI.points}, \text{status} \rightarrow \text{completed})))
\]

The final rule 14 defining the action state A11 combines the patterns of 12 and 13 above.

5 Related Work

We encountered three main previous proposals to provide formal semantics to UML AD.

[Börger et al 2000] proposed to provide semantics for AD by mapping AD elements to transition rules of a multiagent ASM, *i.e.*, an Abstract State Machine with extensions for concurrency. An ASM is essentially a finite automaton where transitions are labeled with rules defining its preconditions and effects. ASM rules appear to capture the operational semantics of the AD in a low-level language of imperative flavor.

[Rodrigues 2000] proposed to provide semantics for an AD by mapping it to a Labelled Transition System (LTS) in two steps, through an intermediate representation called a Finite State Process (FSP). [Eshuis and Wieringa 2001] proposed AD semantics based on a mapping to LTS in two steps, but through a different intermediate representation called an Activity Hypergraph. One advantage of these last two approaches is the availability of automatic model checkers that take as input an LTS model description, together with some temporal or modal logic description of execution ordering and timing constraints.

These previous proposals have in common to define only operational semantics for AD. In addition, they do not cover object flows nor CD, therefore providing semantics for AD in isolation from their structural context in a UML model. They thus seem more relevant for the use of AD in modeling purely procedural concurrent systems, than for their use in object-oriented software engineering.

Our proposal is different in several ways. First, it provides a model-theoretic and a coinciding proof-theoretic semantics for AD, based on FOHL. As pointed out in [Gupta 1999],
such coinciding semantics unify the flavor of denotational semantics brought about by the model theory with those of both axiomatic and operational semantics brought about by the proof-theory. Second, it provides a formal semantics for both AD and CD, linked together through object flows, which makes it more geared towards object-oriented software engineering.

6 Conclusion

In this paper, we proposed to provide a formal semantics to UML AD and CD by mapping their elements to constructors of CTFL, a non-monotonic, OO extension of FOHL. Through this mapping, the semantics of the UML diagrams derives from the coinciding, sound and refutation-complete proof theory and model theory of CTFL. This semantics presents a number of advantages over previous proposals. Foremost, it makes possible to use a single language to:

1. formalize the structure of various UML diagrams;
2. formalize desired temporal execution properties over them, simple ones directly in CTFL and arbitrary complex ones using an additional Event Calculus [Shanahan 1999] layer that is straightforward to axiomatize on top of CTFL;
3. verify their internal and cross-diagram consistency, completeness and temporal correctness through a combination of theorem proving and model checking, and
4. implement the verified model as executable code.

This multiple purpose, single language approach smoothens the learning curve of integrating formal methods with standard object-oriented development. It also brings into the same fold the fast prototyping convenience of the logic programming paradigm. UML is a high-level, declarative, object-oriented language. Representing its formal semantics in a language like CTFL that is also high-level, declarative and object-oriented rather than in a low-level, procedural, purely behavioral language - as in most previous approaches - greatly simplifies automatic translation of UML diagrams into their formalization. In addition, CTFL is both a formal specification language and a general purpose, Turing-complete programming language. Consequently, the verified, formal CTFL semantics of a UML model is already an implementation. This shuts down the major loophole of dual language formal development, one for formal specification, and a different one for implementation, namely that programming errors can easily be introduced during the implementation of a verified model.

In future work, we intend to use the semantics defined in this paper to verify UML models consisting of CD and sequential AD. To that effect, we intend to use the STFL theorem prover Flora that runs on top of the deduction engine XSB. In that perspective, it is interesting to mention the XMC model checker [Ramakrishna et al. 1997] that is also implemented on top of the XSB platform. XMC verifies concurrent systems specified in a CCS-based modeling language with respect to desired temporal properties specified in the modal μ-calculus. The performance of XMC has proven comparable on a set of benchmarks to the procedural model checkers such SPIN [Holzmann and Peled] and Murphi [Dill].

7 References


