Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 95 (2004) 83–109
www.elsevier.com/locate/entcs

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

Franklin Ramalho[a,1,2], Jacques Robin[b,3] and Ulrich Schiel[a,4]

[a] *Departamento de Sistemas e Computação*
Universidade Federal de Campina Grande
Campina Grande, Brazil

[b] *Centro de Informática*
Universidade Federal de Pernambuco
Recife, Brazil

Abstract

We propose Concurrent Transaction Frame Logic (CTFL) as a language to provide formal semantics to UML activity and class diagrams. CTFL extends first-order Horn logic with object-oriented class hierarchy and object definition terms, and with five new logical connectives that declaratively capture temporal and concurrency constraints on updates and transactions. CTFL has coinciding, sound and refutation complete proof and model theories. CTFL allows using a single language to (1) formally describe the semantics of both activity and class diagrams, (2) verify UML models based on these two diagrams using theorem proving and (3) implement the model as an executable, object-oriented logic program.

Keywords: UML Semantics, Object-Oriented Logic Programming, Concurrent Transaction Logic, Frame Logic.

[1] Currently at Centro de Informática at Universidade Federal de Pernambuco, Brazil. This research was supported by grants from CNPq of the Brazilian Federal Government.
[2] Email: franklin@dsc.ufcg.edu.br
[3] Email: jr@cin.ufpe.br
[4] Email: ulrich@dsc.ufcg.edu.br

doi:10.1016/j.entcs.2004.04.007

1 Introduction

The Unified Modeling Language (UML) [18] 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 [9,6,5,21,8,1,16,30]. 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, proposals 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 mainly to CASE tool developers, with axiomatic semantics better geared towards application designers and denotational semantics better geared towards language designers [10];
- rely on structurally impoverished *imperative or functional* formal languages that do not fit well the structure rich object-oriented paradigm used in most UML-based development processes;
- rely on *low-level*, and often quite arcane formal languages [26] 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 [27];
- 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 the 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 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 non-monotonic variant of First-Order Horn Logic (FOHL). Although this approach has the potential to provide semantics and CASE tools for the whole of UML, in this paper, we present a proposal focused on the formal semantics of an activity diagram contextualized by a class diagram [5].

We show how Concurrent Transaction Frame Logic (CTFL) [14] [4] can provide formal semantics for both activity and class diagrams. CTFL is the straightforward integration of two orthogonal yet synergetic extensions of FOHL:

- *Frame Logic (FL)*, an object-oriented extension dealing with complex structural modeling with inheritance hierarchies,
- *Concurrent Transaction Logic (CTL)*, a non-monotonic extension dealing with complex behavioral modeling with concurrent logical database updates, transactions, process communication and temporal execution constraints.

Our approach is based on a mapping between the elements of UML activity and class diagrams and the constructors of CTFL. Through this mapping, these UML diagrams are given proof theoretical and model theoretical formal semantics: 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 elements of UML activity and class diagrams, illustrating each of them on a simple example model. In section 3, we review the object-oriented 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.

[5] We do not cover here the whole complexity of class diagrams, leaving this topic for a separate publication. We focus instead on the main features of class diagrams that are relevant to provide context to activity diagrams.

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 relationships. An illustrative class diagram example is given in Figure 1. It is an extension of the Royal & Loyal (R&L) company information system class diagram presented in [31]. R&L manages fidelity programs for various companies, offering regular customers diverse bonuses such as air miles or discount points. A class diagram 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 types, and the *methods* used to represent the behavior of these objects together with constraints on the type of their input parameters and return values. For example in Figure 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 an `integer` returning method `age`().

A class diagram 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 simpler 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 Figure 1, `Earning` and `Burning` transactions are defined as subclasses of the general `Transaction` class, and each member of this class is associated to one member of the `CustomerCard` class.

What a class diagram does *not* specify is the behavior encapsulated in the methods of the classes. UML provides various other diagrams to that effect. A *State Diagram* is essentially a graph that represents 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 invocating its methods and that of other objects. It specifies the conditions that trigger such change and the resulting, new values. In contrast, an activity diagram is essentially a flowchart, and 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* [22]. 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 object-oriented development, each use-case must in the end be implemented by one method of some class. An activity diagram can also be used to describe the behavioral decomposition

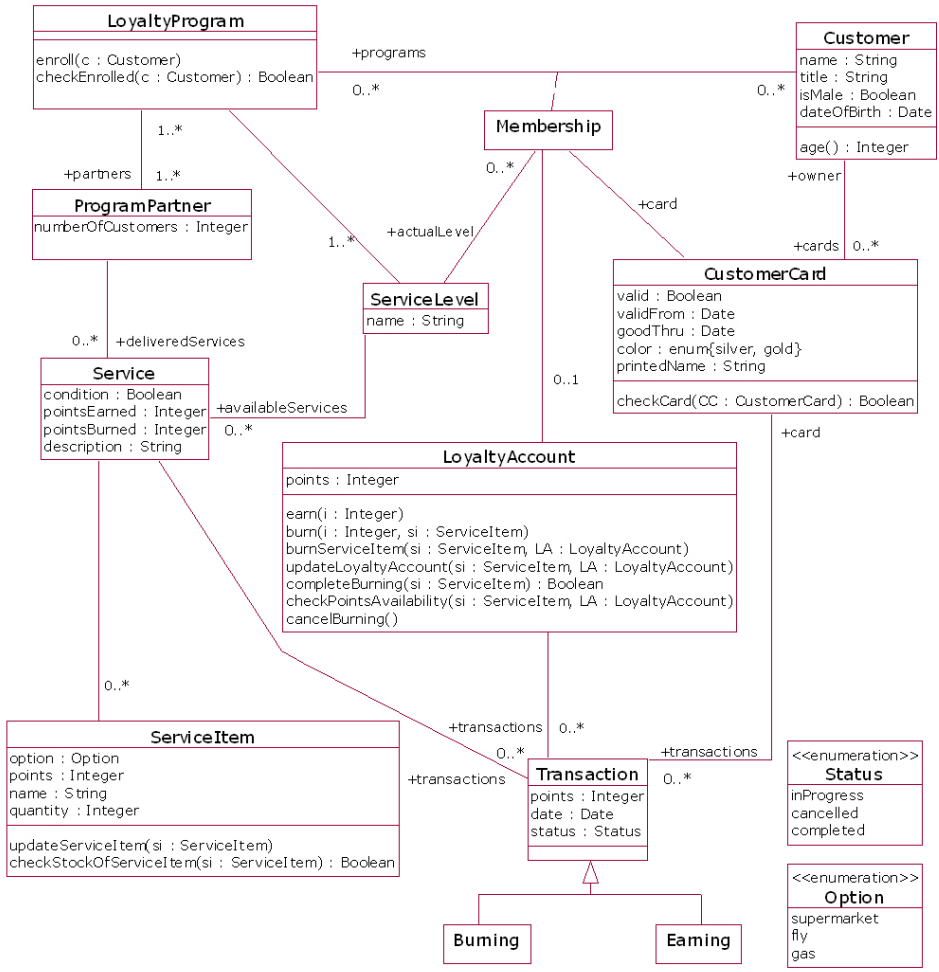


Fig. 1. An example of UML class diagram

and control flow of complex methods implemented by way of invoking methods of objects from various other classes [28]. Although all UML diagrams are useful and complementary for complex system development, use-case, class and activity diagrams can be viewed as the minimal core of UML with which simple object-oriented systems can be specified and modeled. This is why we chose activity and class diagrams as the initial focus of our research on a simple and practical UML model formal semantics.

An illustrative activity diagram example is given in Figure 2. It models the realization of the `burn` method of the `LoyaltyAccount` class from the

class diagram of Figure 1. This method itself realizes the use-case of the same name in the R&L system requirement document. An activity diagram is a graph where nodes are *activities or control constructs* and arcs represents *transitions* between them. Activities are decomposed into atomic *action*

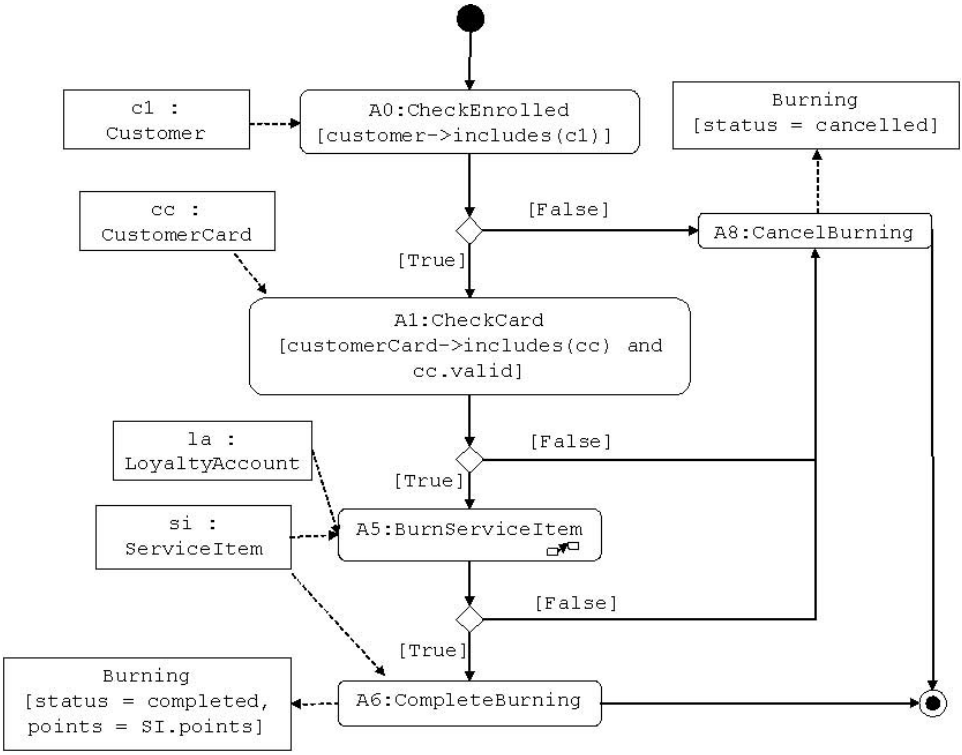


Fig. 2. UML activity diagram modeling the `burn` method of the `LoyaltyAccount` class in the class diagram of Figure 1

states than can be neither decomposed nor interrupted, and *activity states* than can be interrupted and further decomposed into sub-activities. Such decomposition can then be represented by another finer-grained activity diagram. A complex activity can thus be modeled by a hierarchy of activity diagrams, linked to one another through activity states. In addition to its name, an action state can also contain a specification of the operation that it executes. Such specification can be precisely written using the Object Constraint Language (OCL) a textual annotation language, part of the UML standard, that incorporates most basic constructs of logic and algorithms in an intuitive syntax [31]. In the activity diagram of Figure 2, the `BurnServiceItem` node is an example of activity state which behavior is specified by another

activity diagram (shown in Figure 3). All the other nodes of this activity diagram are examples of either action states or control constructs. Activity states can also include *entry* and *exit* actions to be executed immediately before entering and immediately after leaving the state (respectively). The

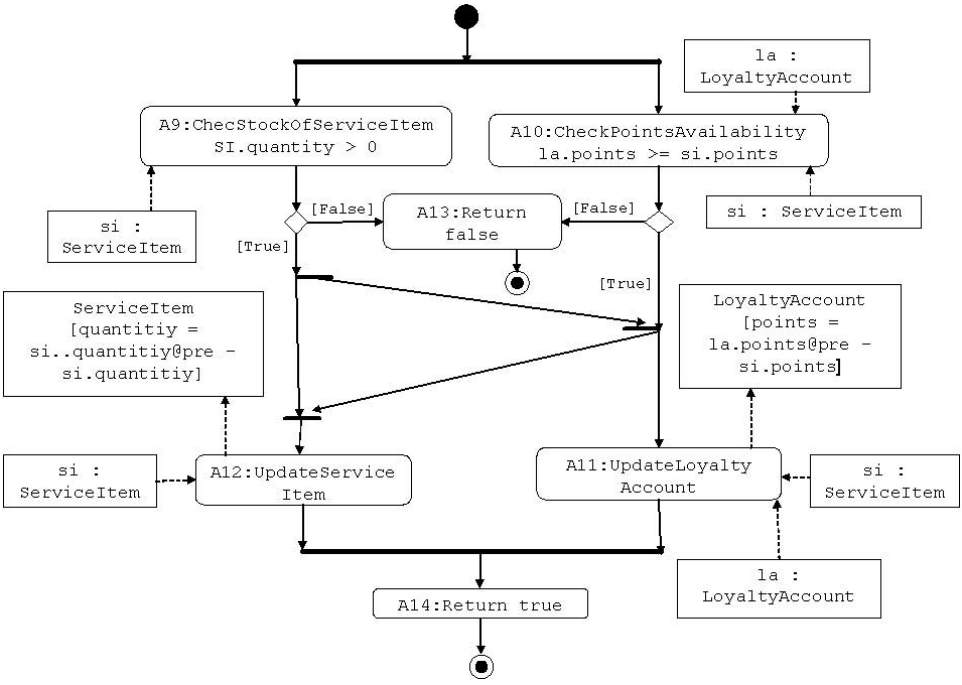


Fig. 3. UML sub-activity diagram for `BurnServiceItem` activity state

control constructs of an activity diagram are: (1) *if/merge* pairs, that represent conditional branching to mutually exclusive threads, (2) *fork/join* pairs, that represent concurrent threads, and (3) *synch states*, that represent inter-thread synchronization constraints. For example in the diagram of Figure 2, the branching node below the `CheckEnrolled` action node models that *either* the action `CheckCard` *or* the action `CancelBurning` must immediately follow. In contrast, the fork node at the top of the diagram of Figure 3, models that the `CheckStockOfServiceItem` and `CheckPointsAvailability` actions must *both* always concurrently follow the positive completion of the `CheckCard` action. In the same diagram, the synch state above the `UpdateLoyaltyAccount` action models specifies that its execution must wait for the positive completion of `CheckStockServiceItem` in the other concurrent thread.

Transitions arcs can be labeled by an *event* whose occurrence triggers the transition from one state to the next or by a *guard*, *i.e.*, a pre-condition that

must be verified for the transition to occur. Both events and guards can be precisely modeled with OCL expressions.

An activity diagram can also include *object flows* that link it to a related class diagram. An object flow associates an action or activity state to a class. An incoming flow specifies the class of the objects that the action expects as input parameters. A simple outgoing flow specifies the class of the object returned as output parameter. An outgoing flow with side effects specifies the class of the objects whose attributes are altered by the execution of the action or activity. Which attributes are altered (and how) can be precisely modeled with OCL expressions. For example in Figure 3, two object flows model that the `UpdateLoyaltyAccount` action takes objects of the classes `ServiceItem` and `LoyaltyAccount` as input parameters and a third one models that the input parameter of class `LoyaltyAccount` has its `points` attribute altered by that action.

3 Concurrent Transaction Frame Logic (CTFL)

CTFL is the integration of FL and CTL, two orthogonal extensions of FOHL, the subset of classical first-order logic where all formulas are in implicative normal form [23] with only one conclusion in each implication. A FOHL formula (also called a logic program) is thus a conjunction of implicitly universally quantified implications, each one either:

- A definite clause of the form c $\leftarrow p_1 \wedge ... \wedge p_n.$, where c, p_1, ..., p_n are positive literals;
- A fact of the form c $\leftarrow$ true, where c is a positive literal - usually abbreviated as c $\leftarrow$.

3.1 Frame Logic (FL)

FL extends first-order Horn logic 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:

$$\texttt{class} :: \texttt{superclass}[...\texttt{attr}_\texttt{i}\texttt{typOp}_\texttt{i}\texttt{type}_\texttt{i}, ..., \texttt{meth}_\texttt{j}(..., \texttt{param}_\texttt{j}^\texttt{k}, ...)\texttt{typOp}_\texttt{j}\texttt{type}_\texttt{j}...]$$

There are four typing operators in FL that instantiate the $\texttt{typOp}_\texttt{n}$ in the above pattern: $*{=}{>}$, $*{=}{>}{>}$, $=>$ and $=>>$. The presence or absence of the $*$ prefix distinguishes between inheritable and non-inheritable type constraints, whereas the $>$ and $>>$ suffixes indicates whether the attribute or method 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:

$$\texttt{object} : \texttt{class}[...\texttt{attr}_i\texttt{assignOp}_1\texttt{value}_i, ..., \texttt{meth}_j(..., \texttt{param}_j^k, ...)\texttt{assignOp}_j\texttt{value}_j...]$$

There are four value assignment operators, that instantiate the $\texttt{assignOp}_n$ 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 object-oriented languages. A method is executed when its return result logical variable unifies with a value during theorem proving. The only difference between attributes and methods is thus that a method can take parameters.

FL class definition and object creation terms are called *F-Molecules*. Logical variables can appear in any position inside these molecules: as object name, class name, attribute name, method name, attribute value, method value or method 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 FOHL literals, which guarantees that semantically, FL remains a first-order logic [32].

In order to illustrate FL more concretely, Figure 4 shows the FL facts representing the R&L class diagram of Figure 1. Facts 3 and 10 define `loyaltyAccount` and `transaction` as top-level classes. In these facts the `: superclass` element of the pattern is simply omitted. Facts 11 and 12 define `burning` and `earning` as two subclasses of `transaction`. These four facts also define the type signature constraints on the attributes of these classes, such as `points*` => `integer` in the `loyaltyAccount` class, and on their methods parameters and return value, such as `burn(integer, serviceItem)*` => `void`. These FL facts also define associations through attributes which types are constrained to other classes of the diagram, such as `card*` => `customerCard` in the definition of the `transaction` class.

- **Fact 1:** customer[name *=> string, title *=> string, isMale *=> void, dateOfBirth *=> date, cards *=≫ customerCard, programs *=≫ loyaltyProgram, memberships *=≫ membership, age()*=> integer] ←.
- **Fact 2:** customerCard[valid *=> void, validFrom *=> date, goodThru *=> date, color *=> {silver; gold}, printedName *=> string, owner *=> customer, membership *=> membership, transactions *=≫ transaction, checkCard(customerCard) *=> void] ←.
- **Fact 3:** loyaltyAccount[points *=> integer, membership *=> membership, transactions *=≫ transaction, earn(integer) *=> void, burn(integer, serviceItem)*=> void, updateLoyaltyAccount(serviceItem, loyaltyAccount) *=> void, cancelBurning() *=> void, completeBurning(serviceItem) *=> void, burnServiceItem(serviceItem, loyaltyAccount) *=> void, checkPointsAvailability(serviceItem, loyaltyAccount) *=> void]←.
- **Fact 4:** loyaltyProgram[customers *=≫ customer, memberships *=≫ membership, serviceLevel(integer) *=>> serviceLevel, partners *=≫ programPartner, enroll(customer) *=> void, checkEnrolled(customer) *=> void] ←.
- **Fact 5:** membership[loyaltyAccount*=> loyaltyAccount, actualLevel *=> serviceLevel, card *=> customerCard, program *=> loyaltyProgram, customer *=> customer] ←.
- **Fact 6:** programPartner[numberOfCustomers*=> integer, loyaltyPrograms *=≫ loyaltyProgram, deliveredServices *=≫ service] ←.
- **Fact 7:** service[condition *=> void, pointsEarned *=> integer, pointsBurned *=> integer, description *=> string, programPartner *=> programPartner, serviceLevel *=> serviceLevel, serviceItem *=≫ serviceItem, transactions *=≫ transaction] ←.
- **Fact 8:** serviceLevel[name *=> string, loyaltyProgram *=> loyaltyprogram, membership *=≫ membership, availableServices *=≫ service] ←.
- **Fact 9:** serviceItem[option *=> {supermarket; fly; gas}, points *=> integer, name *=> string, quantity *=> integer, updateServiceItem(serviceItem) *=> void, checkStockOfServiceItem(serviceItem) *=> void] ←.
- **Fact 10:** transaction[points *=> integer, date *=> date, status *=> {inProgress; cancelled; completed}, card *=> customerCard, loyaltyAccount *=> loyaltyAccount, service *=> service] ←.
- **Fact 11:** burning::transaction[]←.
- **Fact 12:** earning::transaction[]←.

Fig. 4. CTFL fact base giving formal semantics and implementing CD of Figure 1

A pair of proof and model theories of FL is given [13]. The proof theory consists of one *isaReflexivity* axiom, three inference rules for FOHL with equality, *resolution*, *factoring* and *paramodulation*, and nine new inference rules covering the object-oriented semantics: *isaTransitivity*, *isaAcyclicity*,

subclassInclusion, *typeInheritance*, *inputRestriction*, *outputRestriction*, *scalarity*, *merging* and *elimination*. The model theory 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.

3.2 Concurrent Transaction Logic (CTL)

Sequential Transaction Logic (STL) extends FOHL with two new transactional connectives: n-ary *serial conjunction* $\otimes$, and n-ary *serial disjunction* $\oplus$.

Concurrent Transaction Logic (CTL) further extends STL with three additional ones: n-ary *concurrent conjunction* $|$, n-ary *concurrent disjunction* ϑ and unary *atomic modality* $\odot$. These five connectives allow representing in a purely declarative and logical way ordering and synchronization constraints on the execution order of logical proof steps. They provide declarative proof-theoretic and model-theoretic semantics to logic programs and database updates and transactions, as well as to multi-agent and inter-process 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 $\otimes$ 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 $\otimes$ q; if they both succeeded, so does p $\otimes$ q.
- The semantics of a serial disjunction p $\oplus$ q is the negation of p $\otimes$ q: *first* execute p, *then* irrespective of the result, execute q; if either of the two execution succeeded, so does p $\oplus$ q; if they both failed, so does p $\oplus$ q.
- The semantics of a concurrent conjunction p $|$ q is: concurrently execute both p and q. If either of the two executions failed, so does p $|$ q. If they both succeeded, so does p $|$ q.
- The semantics of a concurrent disjunction p ϑ q is the negation of p $|$ q: concurrently execute both p and q; if either of the two execution succeeded, so does p ϑ q; if they both failed, so does p ϑ q.
- In this context, the semantics of classical conjunction p $\wedge$ q becomes: execute both p and q, either concurrently or sequentially in any order; if both succeeded, so does p $\wedge$ q; if either one failed, so does p $\wedge$ q.
- Similary, the semantics of classical disjunction p $\vee$ q remains the negation of the classical conjunction p $\wedge$ q: execute both p and q, either concurrently or sequentially in any order; if both failed, so does p $\vee$ q; if either one succeeded, so does p $\vee$ q.

The truth tables of these sequential and concurrent conjunctions (respectively disjunctions) are identical to that of classical conjunction (respectively disjunction). The difference between these new connectives and their classical counterparts lies only in their execution order constraints: specified and sequential for $\otimes$ and $\oplus$, concurrent for | and ϑ, and unspecified for $\wedge$ and $\vee$. Thus, whereas |, ϑ, $\wedge$ and $\vee$ are commutative, $\otimes$ and $\oplus$ are not.

The atomic modality connective $\odot$, prevents the formula within its scope to be partially executed. If one element of an atomic conjunction scoped by $\odot$ 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 $\odot$(p $\otimes$ q), then all the state changes resulting from the execution of p must be rolled back.

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 nor synchronization primitives in itself. To be used in practice, it must thus be parameterized with a set of such primitives. Atomic change primitives useful for our purpose are insertion and deletion of logical facts in a logical database. Synchronization primitives useful for our purpose are sending and receiving synchronization messages across channels shared by several threads. Thus, CTL({insert(Fact), delete(Fact), send(Channel,Fact), receive(Channel, Fact)}) provides a fully declarative formal semantics for non-monotonic FOHL and database with concurrent updates and transactions.

A pair of coinciding, sound and refutation complete proof and model theories of STL are given in [3]. Their respective extensions to CTL are given in [4]. The model theory is based on a *multi-path* structure that captures the possible states that a logical database can go through when complex transactions are applied to it. These transactions use the classical and transactional connectives of CTL to combine primitive updates. The proof-theory relies on one axiom that states *database invariance* through the application of the *empty transaction*, together with four inference rules for *transaction definition application*, *database query*, *database primitive update* and *atomic transaction execution*.

3.3 Integrating Frame Logic with Transaction Logic

Given that FL extends FOHL by introducing new *terms* and STL and CTL extends it by introducing new *connectives*, these extensions are orthogonal and can be straightforwardly combined, respectively yielding STFL and CTFL. To be precise, it is CTFL({insert(Fact), delete(Fact), send(Channel, Fact), receive(Channel, Fact)}) that we propose as a formal language for UML ac-

tivity and class diagram semantics.

While there is no currently available compiler for CTFL, execution platforms are available for two of its subsets: (1) Flora [32], compiles and efficiently executes STFL programs, and (2) CTR[6] interprets CTL programs. Both these platform are implemented as layers on top of the tabled deductive engine XSB [24], a variant of Prolog that relies on an alternative resolution-based FOHL theorem proving procedure called SLG. This procedure makes XSB both far more declarative and efficient than standard Prolog. It implements the well-founded semantics [29] for negation as failure and it caches partial proof results to avoid both the inefficient redundant computation and the left-recursion termination problems of standard Prolog.

To visually summarize the relationships among the CTFL formalisms and tools and the UML activity diagram and class diagram, we give a UML metamodel [18] of our approach in Figure 5.

4 Mapping a UML class diagram and set of activity diagrams to a CTFL program

In this section, we present our mapping of UML activity diagram and class diagram elements to the CTFL constructs. A CTFL class definition fact base gives the semantics of the class diagram. A CTFL rule base gives the semantics of the activity diagram. In what follows, we associate a generic pattern of each main class diagram and activity diagram element with the corresponding CTFL construct pattern that defines its formal semantics in our proposal.

4.1 Mapping a UML class diagram to a CTFL class definition fact base

(1) Class mapping

A UML class signature is mapped directly onto a FL class definition term as shown in Figure 6. Attribute and methods with Boolean values in UML, do not possess any return value in FL. This is because, every FL attribute or method implicitly has two results: its logical truth value, which is Boolean and its object identifier.

[6] http://www.cs.toronto.edu/ bonner/

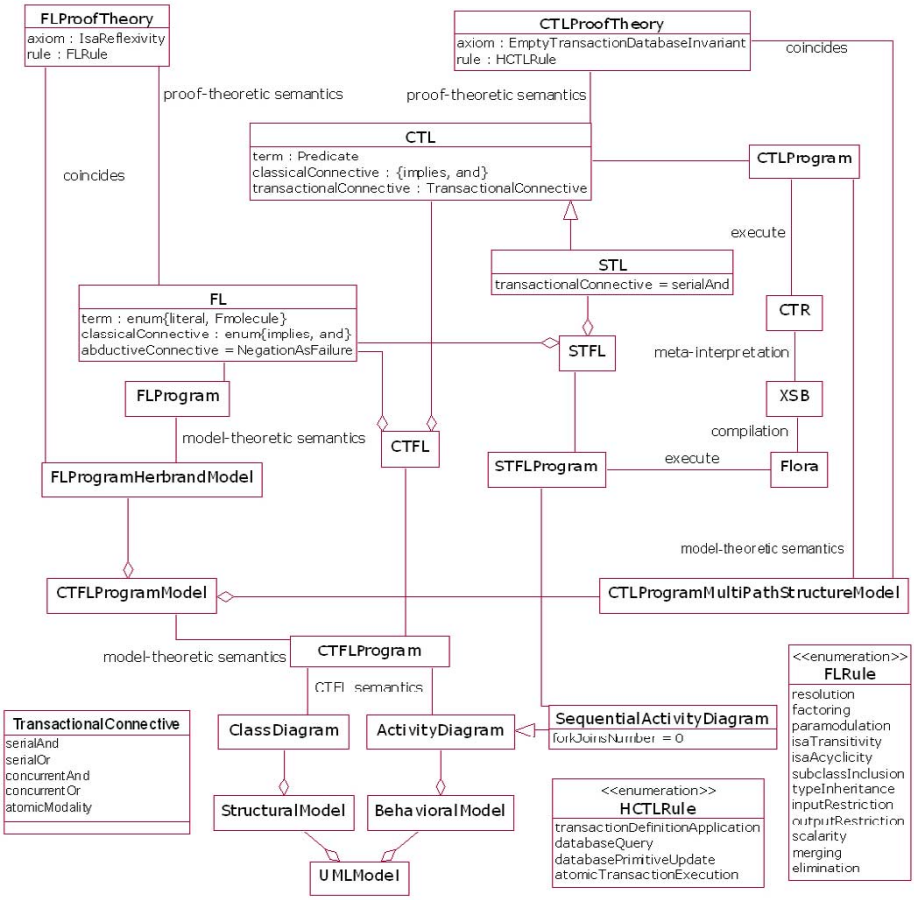


Fig. 5. Meta-Model of our CTFL semantics proposal for UML AD and CD

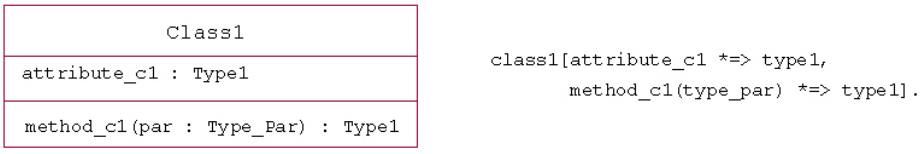
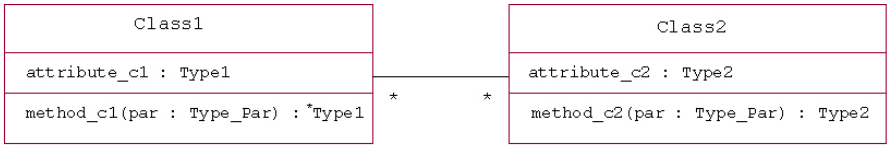


Fig. 6. Class mapping

(2) Association mapping

A UML association is mapped onto FL attributes of the associated classes, following the multiplicity constraints. For example, in Figure 7, `class1` has a set valued attribute referencing `classe2` and vice-versa.



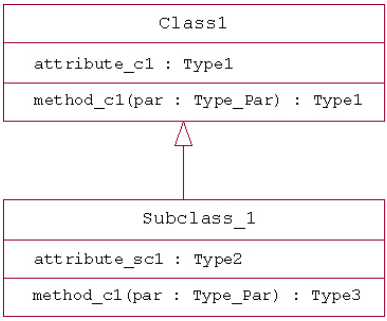

```
class1[attribute_c1 *=> type1,
       method_c1(type_par)* => type1,
       class2 *=>> class2].
```

```
class2[attribute_c2 *=> type2,
       method_c2(type_par)* => type2,
       class1 *=>> class1].
```

Fig. 7. Association mapping

(3) Specialization mapping

A UML specialization relationship is mapped onto a FL subclass "::" operator, as in the second F-Molecule of Figure 8. The default inheritability of UML attributes and methods is mapped onto the type constraint operators prefixed by * that captures such semantics in FL. Examples from the R&L case study of these three mappings above were discussed in section 3.1.




```
class1[attribute_c1*=> type1,
       method_c1(type_par) *=> type1].
subclass_1::class1[attribute_sc1 *=> type2,
                   method_c1(type_par) *=> type3].
```

Fig. 8. Inheritance mapping

4.2 Mapping a UML activity diagram to a CTFL Rule Base

Each path in an activity diagram is mapped onto one CTFL clause which conclusion corresponds to the overall activity modeled by the diagram. The nodes and transitions of each path are then mapped onto the premises of the corresponding CTFL clause following the rules below.

(1) Mapping action states

An action state A with no OCL constraint to further specify the behavior of this action is mapped onto a CTFL term (*i.e*, an F-molecule or a predicate with F-molecules as arguments) that appears as premise in each of the clauses that represent the activity diagram paths where A appears. This general mapping is shown in Figure 9. This is the case for example of the action state A8 in Figure 2 onto `LA[cancelBurning()]` term in rules 1-3 in Figure 10. Such mapping is also carried out for actions with OCL constraints. However, in this case, an additional clause is added, with the action as conclusion and the OCL constraints as premises [7] .

Fig. 9. Action state mapping

This is the case for example of the mapping from action state A1 in Figure 2 onto `CC[checkCard(CC)]` term that appears as conclusion of rule 6 and premise in rules 2-4 in Figure 10.

[7] Mapping a logical OCL expression to a CTFL premise is beyond the scope of this paper.

- **Rule 1:** Path ≪A0, A8≫
  LA[burn(Pts, SI)] ← ⊙(LA:loyaltyAccount ⊗
  LA[membership[programs− >>LP:loyaltyProgram,
  customer− >>C1:customer]]
  ⊗ ¬LP[checkEnrolled(C1)] ⊗ LA[cancelBurning()]).
- **Rule 2:** Path ≪A0, A1, A8≫
  LA[burn(Pts, SI)] ← ⊙(LA:loyaltyAccount ⊗
  LA[membership[programs− >>LP:loyaltyProgram,
  customer− >>C1:customer, card− >CC]]
  ⊗ CC[owner− >C1] ⊗ LP[checkEnrolled(C1)] ⊗ ¬CC[checkCard(CC)] ⊗
  LA[cancelBurning()]).
- **Rule 3:** Path ≪A0, A1, A5, A8≫
  LA[burn(Pts, SI)] ← ⊙(LA:loyaltyAccount ⊗
  LA[membership[programs− >>LP:loyaltyProgram,
  customer− >>C1:customer, card− >CC]]
  ⊗ CC[owner− >C1] ⊗ LP[checkEnrolled(C1)] ⊗ CC[checkCard(CC)] ⊗
  SI:serviceItem ⊗ ¬LA[burnServiceItem(SI, LA)] ⊗ LA[cancelBurning()]).
- **Rule 4:** Path ≪A0, A1, A5, A6≫
  LA[burn(Pts, SI)] ← ⊙(LA:loyaltyAccount ⊗
  LA[membership[programs− >>LP:loyaltyProgram,
  customer− >>C1:customer, card− >CC]]
  ⊗ CC[owner− >C1] ⊗ LP[checkEnrolled(C1)] ⊗ CC[checkCard(CC)] ⊗
  SI:serviceItem ⊗ LA[burnServiceItem(SI, LA)] ⊗ LA[completeBurning(SI)]).
- **Rule 5:** state A0
  LP[checkEnrolled(C1)] ← C1:customer ⊗ LP[customer− >>C1].
- **Rule 6:** action state A1
  CC[checkCard(CC)] ← CC:customerCard ⊗ CC.valid.
- **Rule 7:** action state A6
  LA[completeBurning(SI)] ← SI:serviceItem ⊗
  (insert(_:burning[points→SI.points, status→completed])).
- **Rule 8:** action state A8
  LA[cancelBurning()] ← ⊙(insert (:burning[status→cancelled])).

Fig. 10. CTFL rule base giving formal semantics and implementing AD of Figure 2

(2) Mapping activity states

An activity state D is mapped onto one to four CTFL terms that appear as premises in each of the clauses that represent the activity diagram paths where D appears. This general mapping is shown in Figure 11. In the simplest case, where the activity has neither entry condition, nor exit action nor interrupting event, the single premise consist of the CTFL term that contains the invocation of method D. This is the case for example of the mapping from the A5 activity state in Figure 2 onto the ¬`LA[burnServiceItem(SI, LA)]` premise in rule 3 and `LA[burnServiceItem(SI, LA)]` premise in rule 4 of Figure 10.

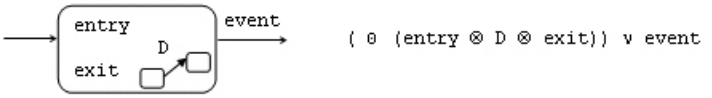


Fig. 11. Activity state mapping

In the most complex case, with all the optional elements present, there is one additional premise per element. Three premises are grouped in a serial conjunction in the following order: the *entry action*, then the complex activity D, and finally the *exit action*. This serial conjunction is surrounded by an atomic modality operator to roll back the side effects of the entry action and the complex activity D if an interruption occurs before the execution of the exit action. This transaction is conjoined with a possibly interrupting event in a concurrent disjunction.

In all cases, the entire activity diagram mapping process is recursively re-applied onto the sub-activity diagram that further specifies the behavior D. This results in the introduction of new clauses in the CTFL program. This is the case for example of the recursive mapping of the activity diagram of Figure 3 onto the rules of Figure 12.

(3)Mapping fork and join pairs

The concurrent nodes between a fork and a join in an activity diagram are mapped directly onto a CFTL concurrent conjunction. This concurrent conjunction is the central element of a serial conjunction that starts with the guard on the transition leading to the fork and ends with the guard out on the transition following the join. This general mapping is shown in Figure 13.

(4) Mapping fork and join pairs with synch states

In a UML activity diagram, synch state bars may be used within concurrent threads to represent synchronized states. These synch states are mapped to the synchronization primitives that parametrize CTFL in our formal semantics proposal. An incoming arc *to* such a synch state *from* another thread is mapped onto a `receive(Channel, done)` premise in each of the CTFL clauses that represent the activity diagram paths where the synch state appears. An outgoing arc *from* such a synch state *to* another thread is mapped onto a `send(Channel, done)` premise in each the CTFL clauses that represent the activity diagram paths where the synch state appears. The `Channel` parameter is used to identify the other thread from which the incoming arc is coming or to where the outgoing arc is going. These `send` and `receive` actions are joint

- **Rule 9:** activity state A5
  LA[burnServiceItem(SI, LA)] ← ⊙((SI:serviceItem ⊗ LA:loyaltyAccount) ⊗
  (SI[checkStockOfServiceItem(SI)] ⊗ send(ch1, done) ⊗
  receive(ch2, done) ⊗ SI[updateServiceItem(SI)])
  | (LA[checkPointsAvailability(SI, LA)] ⊗ receive(ch1, done) ⊗
  send(ch2, done) ⊗ LA[updateLoyaltyAccount(SI, LA)])).
- **Rule 10:** activity state A5
  LA[burnServiceItem(SI, LA)] ← ⊙((SI:serviceItem ⊗ LA:loyaltyAccount) ⊗
  (¬SI[checkStockOfServiceItem(SI)])
  | (LA[checkPointsAvailability(SI, LA)] ⊗ receive(ch1, done) ⊗
  send(ch2, done) ⊗ LA[updateLoyaltyAccount(SI, LA)])).
- **Rule 11:** activity state A5
  LA[burnServiceItem(SI, LA)] ← ⊙((SI:serviceItem ⊗ LA:loyaltyAccount) ⊗
  (SI[checkStockOfServiceItem(SI)] ⊗ send(ch1, done) ⊗
  receive(ch2, done) ⊗ SI[updateServiceItem(SI)])
  | (¬LA[checkPointsAvailability(SI, LA)])).
- **Rule 12:** activity state A5
  LA[burnServiceItem(SI, LA)] ← ⊙((SI:serviceItem ⊗ LA:loyaltyAccount) ⊗
  (¬SI[checkStockOfServiceItem(SI)])
  | (¬LA[checkPointsAvailability(SI, LA)])).
- **Rule 13:** action state A9
  SI[checkStockOfServiceItem(SI)] ← SI:serviceItem ⊗ SI.quantity > 0.
- **Rule 14:** action state A10
  LA[checkPointsAvailability(SI, LA)] ← LA.points > SI.points.
- **Rule 15:** action state A11
  LA[updateLoyaltyAccount(SI, LA)] ← ⊙(SI:serviceItem ⊗ LA:loyaltyAccount ⊗
  Pre = LA.points ⊗ delete(LA.points) ⊗ insert(LA[points→($Pre - SI.points$)]
  )).
- **Rule 16:** action state A12
  SI[updateServiceItem(SI)] ← ⊙(SI:serviceItem ⊗ LA:loyaltyAccount ⊗
  Pre = SI.quantity ⊗ delete(SI.quantity) ⊗ insert(SI[quantity→($Pre - 1$)])).

Fig. 12. CTFL rule base giving formal semantics and implementing AD of Figure 3

in serial conjunctions with the other actions of the thread where the synch state occurs. This general mapping is shown in Figure 14.

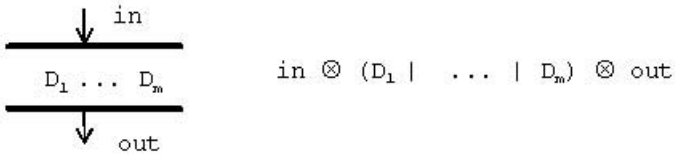
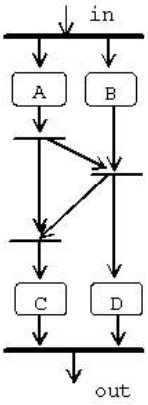


Fig. 13. Fork and join control flow mapping

An example of a bi-lateral synchronization between two threads is given in the activity diagram of Figure 3. One thread includes action states A9 and A12, while the other concurrent thread includes action states A10 and




```
in ⊗ (A ⊗ send(ch1, done) ⊗ receive (ch2, done) ⊗ C) |
 (B ⊗ receive (ch1, done) ⊗ send(ch,2 done) ⊗ D) ⊗ out
```

Fig. 14. Fork and join control flow with synch states mapping

A11. These two threads are synchronized by two arcs between three synch states. The arc that goes out from the top synch state in the A9-A12 thread to the top synch state in A10-A11 thread forces the execution of A11 to wait for the positive completion of A9 in addition to the positive completion of A10. It is mapped onto the synchronization predicates `send(ch1, done)` and `receive(ch2, done)` in the premise of CTFL rule 9 of Figure 12. Similarly, the arc that goes out from the synch state in the A10-A11 thread to the bottom synch state in the A9-A12 thread forces the execution of A12 to wait for the positive completion of A10 in addition to the positive completion of A9. It is mapped onto the synchronization predicates `receive(ch1, done)` and `send(ch2, done)` in the CTFL rule 10 of Figure 12.

(5) Mapping branching nodes

A UML activity diagram can contain two different kinds of branching nodes: (1) Boolean ones, with two outgoing transitions, one corresponding to the result of the previous activity being true and the other corresponding to the result of that activity being false, and (2) multiple choice ones, with at least three outgoing transitions, each one distinguished by a guard. A sub-branch leading from an activity A to an activity B through a Boolean branching node is mapped onto a serial conjunction in the premise of the clause that represent the activity diagram paths where the branching node appears. If the sub-branch includes the positive outgoing transition of the node, this serial conjunction starts with the CTFL term representing A. If it includes the negative outgoing transition, the serial conjunction starts with the negation of that term. In both cases, the serial conjunction ends with the CTFL term representing B. This is the case for example of the mapping from

the top Boolean branching node in the diagram of Figure 2, onto the serial conjunction ¬LP[checkEnrolled(C1)] ⊗ LA[cancelBurning()] in the premise of the CTFL rule 1 in Figure 10. A sub-branch leading from activity A to activity B through a multiple choice guarded outgoing transition is mapped onto a serial conjunction starting with the CTFL term representing A, followed by the serial conjunction representing the OCL constraint encoding the guard, followed by the CTFL term representing B. These three mappings are illustrated in Figure 15.

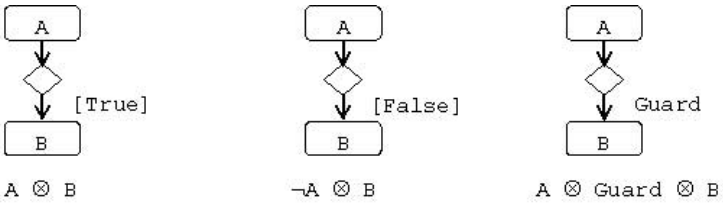


Fig. 15. Branching nodes mapping

4.3 Mapping Object Flows

In UML, object flows link the behavioral activity diagrams to the structural class diagram. They are mapped onto CTFL terms that link the behavioral CTL clauses with the structural FL clauses. Each object flow associated to an action state is mapped onto one or several CTFL terms that appear as additional premises in the rule that resulted from mapping this state using the pattern of Figure 9 that we discussed earlier.

(1) Mapping input parameter object flows

An object flow that specifies that the input parameter of an action state A is of class C is mapped onto a single additional premise of the general form shown in Figure 16. Note that only the boldfaced part of the CTFL rule represents the object flow itself. The rest of the rule results from mapping the action state A using the pattern of Figure 9. Note also that the object variable in the added premise (in bold) must appear in the conclusion of this CTFL rule. An as example of such mapping, consider the top object flow in Figure 2 that specifies that the input parameter of action A0 is an object of class Customer. This flow is mapped onto the CTFL term C1 : customer that appears as an additional premise of rule 5 in Figure 10. The conclusion and other premise of this rule result from mapping action state A0 itself.

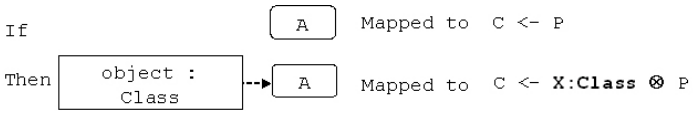


Fig. 16. Input parameter object flow mapping

(2) Mapping object creation object flows

An action state object creation flow is mapped onto a single additional premise of the general form shown in Figure 17 . This premise contains an atomic update primitive that inserts to the CTFL theory an additional anonymous object of the class specified in the flow. The OCL expression specifying the attribute values of the object created by the flow is mapped onto a corresponding CTFL term describing this new object. As an example, consider the object flow outgoing from action state A6 in Figure 2 and the corresponding premise $\texttt{insert}(\texttt{\_:burning}[\texttt{points}- > \texttt{SI.points}, \texttt{status}- > \texttt{completed}])$ in rule 7 of Figure 10.

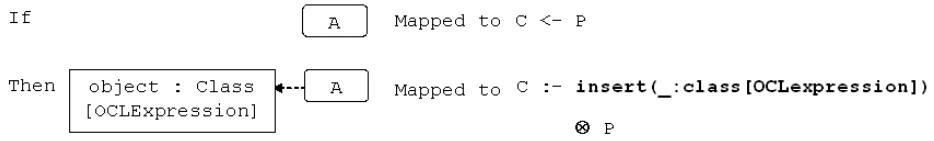


Fig. 17. Object creation object flow mapping

(3) Mapping attribute value update object flows

An action state object attribute alteration flow is mapped onto an atomic serial conjunction of three additional premises. The first element is a CTFL term that relates the new value to the old one. The second element is a primitive CTFL theory update that retracts the old, obsolete value of the attribute. The third element is another such primitive that inserts the new value. Each of these terms take as argument the corresponding subexpression in the OCL constraint specified in the object flow. The general for such mapping is given in Figure 18. As an example, consider the object flow outgoing from action state A12 in Figure 3 and the corresponding additional three premises $\texttt{New} = \texttt{SI.quantity} - 1 \otimes \texttt{delete}(\texttt{SI.quantity}) \otimes \texttt{insert}(\texttt{SI}[\texttt{quantity}- > \texttt{New}])$ in rule 16 of Figure 12.

Applying the general mappings defined in Figures 6-9, 11, and 13-18 above to the particular class diagram of Figure 1 and activity diagrams of Figures 2 and 3 yields the CTFL program shown in Figures 4, 10 and 12. This CTFL

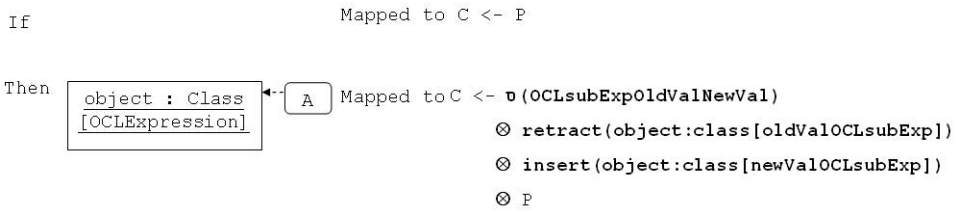


Fig. 18. Object creation object flow mapping

program represents both the formal semantics of the three UML diagram and their implementation as an executable object-oriented logic program.

In our mapping, we do not consider the following UML activity diagrams constructs: swimlanes, deferred events. Swimlanes correspond to organizational units in a business model that are used to organize responsibility for action and sub-activities. They *do not impact the execution semantics*. Deferring an event e can be simulated by using the guard [e occurred] [8].

5 Related Work

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

The first [5] proposed a semantics based on a mapping of activity diagram elements onto 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 an activity diagram in a low-level language of imperative flavor.

The second [21] proposed a semantics based on mapping an activity diagram onto a Labelled Transition System (LTS) in two steps, through an intermediate representation called a *Finite State Process* (FSP). The third [8] also proposed a semantics based on a mapping to an 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 the *operational* semantics for activity diagrams. In addition, they do not cover object flows nor class diagrams, therefore providing semantics for activity diagrams *in isolation from their structural context* in a UML model. They thus seem more relevant

for the use of activity diagrams in modeling purely imperative concurrent systems, than for their use in object-oriented software engineering.

Our proposal is different in two ways. First, it provides a *model-theoretic* and a coinciding *proof-theoretic* semantics for activity diagrams, based on a non-monotonic extension of FOHL. As pointed out in [10], semantics based on such logic 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, our proposal provides a formal semantics for both *activity* and *class* diagrams, linked together through object flows, which makes it more geared towards object-oriented software engineering.

6 Conclusion

In this paper, we proposed to provide formal semantics to UML activity and class diagrams by mapping their elements to constructors of CTFL, a non-monotonic, object-oriented extension of first-order Horn logic. 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:

(i) Formalize the structure of various UML diagrams;

(ii) Formalize desired temporal execution properties over them, simple ones directly in CTFL and arbitrary complex ones using an additional Event Calculus [25] layer that is straightforward to axiomatize on top of CTFL;

(iii) Verify their internal and cross-diagram consistency, completeness and temporal correctness through a combination of theorem proving and model checking;

(iv) 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 mapping presented in this paper in the MODELOG model-driven formal software development framework. Its architecture is shown in Figure 19.

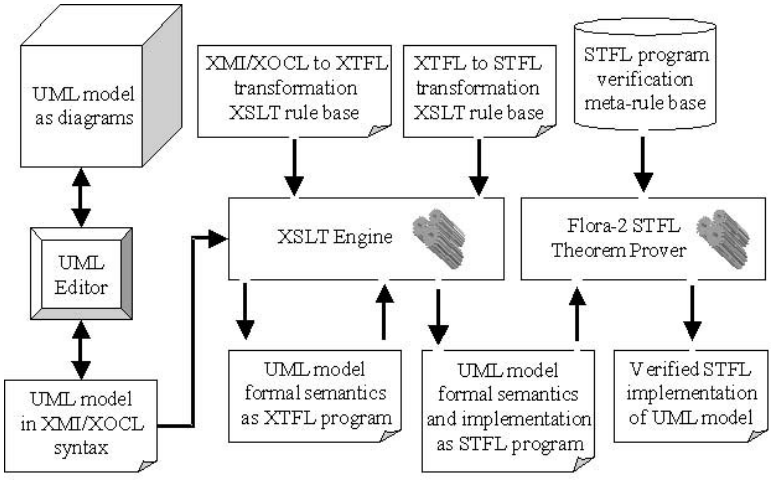


Fig. 19. MODELOG architecture

The envisioned development process that MODELOG will support is the following. It starts by drawing a UML model using a UML editor. This model is then exported in XML syntax using (1) the standard XMI format [11] for the diagrammatic part of the model and (2) the XOCL format [20] that we developed for the OCL part. A set of XSLT transformation rules [2] that capture the mapping described in this paper are then applied to automatically generate from this XMI/XOCL representation of the model a corresponding XTFL representation. XTFL is an XML format for CTFL that we developed. A second set of XSLT transformation rules then maps this XTFL representation onto the STFL syntax accepted as input by the Flora-2 theorem prover. This prover is first used to verify the consistency and completeness of the automatically generated STFL program. To carry out this task, the prover simply applies to this STFL program seen as data, a meta-level verification query and rule base implemented in STFL itself. Missing or inconsistent elements discovered during verification points to modeling problems. The developer then goes back to the drawing board, corrects the problems and reapplies the transformation pipeline. Once verified correct, the STFL program - together

with the Flora-2 prover now serving as a run-time execution object-oriented logic programming platform - constitutes the automatically generated implementation of the UML model. In this perspective, it is interesting to mention the XMC model checker [19], which, like Flora-2, is also implemented on top of XSB. XMC verifies concurrent systems specified in a CCS-based [17] modeling language with respect to desired temporal properties specified in the modal μ-calculus [15]. The performance of XMC has proven comparable on a set of benchmarks to procedural model checkers such as SPIN [12] and Murphi [7].

References

[1] Aredo, D. B. *A Framework for Semantics of UML Sequence Diagrams in PVS*, Journal of Universal Computer Science (JUCS), 8(7), (2002), pp. 674-697, July.

[2] Birbeck, M., Duckett, J., Gudmundsson, O. G., Kobalo, P., Lens, E., Livingstone, S., Marcus, D., Mohr, S., Pinnock, J., Visco, K., Watt, A., Williams, K., Zaev, Z. and Ozu, N., "Professional XML", Wrox Press Ltd, Birmingham, 2000.

[3] Bonner, A. J. and Kifer, M. "Transaction logic programming, a logic for procedural and declarative knowledg", Technical Report CSRI-323, Computer Systems Research Institute,University of Toronto, 1995.

[4] Bonner, A.J. and Kifer, M. *Concurrency and communication in transaction logic.* In Proceedings of the Joint International Conference and Symposium on Logic Programming, (1996), pp. 142-156. MIT Press.

[5] Börger, E., Cavarra, A. and Riccobene, E. *An ASM semantics for UML activity diagrams.* Algebraic Methodology and Software Technology (AMAST 2000), T. Rus, volume 1816 of Lecture Notes in Computer Science, (2000), Springer.

[6] DeLoach, S. Hartrum, T. and Smith, J. "A Theory-Based Representation for Object-Oriented Domain Models", IEEE Transactions on Software Engineering, Volume 26 no. 6, (1999), June.

[7] Dill, D. L. "he Murphi verification system", Computer Aided Verification (CAV'96), volume 1102 of Lecture Notes in Computer Science, New Brunswick, New Jersey, July, Spring-Berlag, (1996), pages 390-393.

[8] Eshuis, R. and Wieringa, R. "A formal semantics for UML activity diagrams -Formalising workflow models". Technical Report CTIT-01-04, University of Twente, Department of Computer Science, 2001.

[9] Gogolla, M. and Parisi-Presicce, F. *State diagrams in UML: A formal semantics using graph transformations.* ICSE'98 Workshop Precise Semantics of Modeling Techniques, Manfred Broy, Derek Coleman, Tom Maibaum, and Bernhard Rumpe, Technical Report TUM-I9803, (1998), pages 55-72.

[10] Gupta, G. "Horn logic denotations and their applications", The Logic Programming Paradigm: A 25 year perspective, Springer-Verlag, 1999.

[11] Grose, T., Doney, G.C. and Brodsky, S.A. "Mastering XMI: Java Programming with XMI, XML and UML". john Wiley and sons, 2002.

[12] Holzmann, G. J. and Peled, D. "The state of SPIN", Computer Aided Verification (CAV'96), volume 1102 of Lecture Notes in Computer Science, New Brunswick, New Jersey, July, Spring-Berlag, (1996), pages 385-389.

[13] Kifer, M. Lausen, G. and Wu, J. "Logical foundations of object-oriented and frame-based languages", Journal of the ACM, pp.741-843. 1995.

[14] Kifer, M. *Deductive and Object Data Languages: A Quest for Integration*, 4th International Conference on Deductive and Object-Oriented Databases, Singapore, December, Lecture Notes in Computer Science, no. 1013, (1995), Springer Verlag, New York.

[15] Kozen, D. "Results on the propositional μ-calculus". Theoretical Computer Science, 27:333-354, 1983.

[16] Kuske, S., Gogolla M. Kollmann, R. and Kreowski, J. *An Integrated Semantics for UML Class, Object, and State Diagrams based on Graph Transformation*, 3rd Int. Conf. Integrated Formal Methods (IFM'02), Butler, M. and Sere, K., (2002), Springer.

[17] Milner, R. "Communication and Concurrency". International Series in Computer Science. Prentice Hall, 1989.

[18] OMG (2003) OMG Unified Modeling Language Specification - Version 1.5, March 2003, OMG URL: http://www.omg.uml.com/

[19] Ramakrishna, Y. S., Ramakrishnan, C. R., Ramakrishnan, I. V., Smolka, S. A., Swift, T. W. and Warren, D. S. *Efficient model checking using tabled resolution*, 9th International Conference on Computer-Aided Verification (CAV '97), Haifa, Israel, (1997), July, Springer-Verlag.

[20] Ramalho, F., Robin, J. and Barros, R. S. M. *XOCL - An XML language for specifying logical constraints in object oriented models*, Journal of Universal Computer Science (2003), 9(8), Springer, URL: http://www.jucs.org/jucs_9_8/xocl_an_xml_language.

[21] Rodrigues, R. W. S. *Formalising UML Activity Diagrams using Finite State Processes*. Dynamic Behaviour in UML Models: Semantic Questions, York. UML 2000, 2000.

[22] Rumbaugh, J., Jacobson, I. and Booch, G. "The Unified Modeling Language -Reference Manual", (1999), Addison-Wesley.

[23] Russell, S. and Norvig, P. "Artificial Intelligence: A Modern Approach", Prentice Hall, second edition, 2002.

[24] Sagonas, K., Swift, T. and Warren, D. S. *XSB as an efficient deductive database engine*, Snodgrass, R. T. and M. Winslett, M., ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'94), (1994), pages 442-453.

[25] Shanahan, M. P. "The Event Calculus Explained", Artificial Intelligence Today, Wooldridge, M. J. and Veloso, M. Springer Lecture Notes in Artificial Intelligence no. 1600, (1999), pages 409-430, Springer.

[26] Schmidt, D. A. "On the need for a popular formal semantics". ACM SIGPLAN Notices, (1997), 32(1):115-116.

[27] Schmidt, D.A. *Should UML Be Used for Declarative Programming?* Proc. ACM Conf. on Principles and Practice of Declarative Programming (PPDP'01), 2001.

[28] Stevens, P. and Pooley, S. "Using UML Software Engineering with Objects and Components", Object Techinology, (2000), Addison-Wesley.

[29] Van Gelder, A., Ross, K.A., and Schlipf, J.S. "The Well-Founded Semantics for General Logic Programs". Journal of the ACM 38(3):620–650, 1991.

[30] Varro, D. *A formal semantics of UML Statecharts by model transition systems*. ICGT 2002: International Conference on Graph Transformation, 2002.

[31] Warmer, J. and Kleppe, A. "The object constraint language: precise modeling with UML", Object technology series, Addison-Wesley, 1999.

[32] Yang, G. and Kifer, M. "Flora: Implementing an efficient DOOD system using a tabling logic engine", Lloyd, J., Dahl, V., Furbach, U., Kerber, M., Lau, K., Palamidessi, C., Pereira, L. M., Sagiv, Y. and Stuckey, P. J., Computational Logic - CL-2000, number 1861 in LNAI, pages 1078–1093. Springer, (2000), July.