National Technical University of Athens

School of Electrical and Computer Engineering

Division of Information Transmission Systems and Material Technology

Enabling Reasoning and Verification Support for Intelligent Agent Systems, using Formal Methods

Doctorate Thesis

Aikaterini E. Ksystra

Athens, December 2017

National Technical University of Athens

School of Electrical and Computer Engineering

Division of Information Transmission Systems and Material Technology

Enabling Reasoning and Verification Support for Intelligent Agent Systems, using Formal Methods

Doctorate Thesis

Aikaterini E. Ksystra

Advisory Committee:

Panayiotis Frangos (Thesis Advisor)
Petros Stefaneas (Thesis Committee)
Konstantine Arkoudas (Thesis Committee)

Approved by the Examination Committee on

.....	.....	.....
P. Frangos NTUA	P. Stefaneas NTUA	K. Arkoudas Bloomberg Research
.....	.....	.....
S. Papavassiliou NTUA	G. Koletsos NTUA	K. Dimitrakopoulos University of Athens

.....

P. Kavassalis
University of Aegean

Athens, December 2017

.....
Αικατερίνη, Ε. Ξύστρα

Υποψ. Διδάκτωρ της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Ε.Μ.Π.

Διπλωματούχος της Σχολής Εφαρμοσμένων Μαθηματικών και Φυσικών Επιστημών, Ε.Μ.Π.

Αφιερώνεται στην οικογένεια μου.

Περίληψη

Οι τυπικές μέθοδοι είναι τεχνικές, γλώσσες και εργαλεία με ισχυρό μαθηματικό υπόβαθρο, οι οποίες μας δίνουν τη δυνατότητα μιας αυστηρής, μαθηματικά ορισμένης, χωρίς ασάφειες, περιγραφής ή προδιαγραφής των συστημάτων, η οποία χρησιμοποιείται για το σχεδιασμό τους, την ανάλυση, και την επαλήθευση επιθυμητών ιδιοτήτων τους.

Ένας πολύ σημαντικός κλάδος των τυπικών μεθόδων είναι οι γλώσσες αλγεβρικών προδιαγραφών, οι οποίες έχουν ως μαθηματικό υπόβαθρο κάποιο μαθηματικό λογικό σύστημα ή και συνδυασμούς λογικών συστημάτων. Μια τέτοια εκτελέσιμη αλγεβρική γλώσσα προδιαγραφών νέας γενιάς, η οποία είναι απόγονος των ιστορικών γλωσσών OBJ, είναι η CafeOBJ. Το βασικότερο χαρακτηριστικό της γλώσσας αυτής, που τη διακρίνει από αντίστοιχους φορμαλισμούς, είναι η άμεση υποστήριξη που παρέχει για συμπεριφοριακές προδιαγραφές, με την ενσωμάτωση στο συντακτικό της ειδικών τύπων και τελεστών. Η συμπεριφοριακή προδιαγραφή, η οποία έχει τις βάσεις της στην άλγεβρα με κρυμμένους τύπους, ενισχύει την αλγεβρική προδιαγραφή με αντικείμενα ή αφηρημένες μηχανές καταστάσεων, δίνοντας έτσι τη δυνατότητα περιγραφής σύνθετων δυναμικών συστημάτων. Παράλληλα, με τη χρήση τεχνικών επαλήθευσης που βασίζονται στη συμπεριφοριακή προδιαγραφή, η CafeOBJ μας δίνει τη δυνατότητα απόδειξης ιδιοτήτων ασφαλείας των συστημάτων που έχουν προδιαγραφεί.

Αντικείμενο της διατριβής αποτελεί η μοντελοποίηση και η επαλήθευση της συμπεριφοράς έξυπνων συστημάτων με χρήση τεχνικών αλγεβρικών προδιαγραφών.

Με τον όρο έξυπνα συστήματα (ή έξυπνοι πράκτορες) αναφερόμαστε στα συστήματα τα οποία χρησιμοποιούνται στον Νέο ή Σημασιολογικό Ιστό ώστε να εξασφαλίζεται η ανάγκη για αντιδραστικότητα (reactivity), αλληλεπίδραση και επικοινωνία μεταξύ των διαφόρων συστατικών του. Ένα έξυπνο σύστημα μπορεί να οριστεί σαν μία αυτόνομη οντότητα η οποία παρατηρεί μέσω αισθητήρων το περιβάλλον του και έπειτα με βάση τις παρατηρήσεις δρα κατάλληλα σε αυτό, κατευθύνοντας δηλαδή τις δραστηριότητές του προς την επίτευξη των στόχων του.

Λόγω της ιδιαιτερότητας των συστημάτων αυτών, και εξαιτίας της αύξησης της ανάπτυξής τους καθώς και της χρήσης τους σε κρίσιμα συστήματα, οι απαιτήσεις για αξιοπιστία, ασφάλεια και λειτουργικότητα έχουν οδηγήσει στην ανάγκη για χρήση τυπικών μεθόδων για την ανάλυση τους και στην ανάπτυξη νέων μεθοδολογιών προσανατολισμένων συγκεκριμένα σε αυτά τα συστήματα.

Σε αυτήν την διατριβή προτείνεται ένα πλαίσιο προδιαγραφής και επαλήθευσης δύο βασικών κατηγοριών έξυπνων πρακτόρων, εκείνων των συστημάτων των οποίων η συμπεριφορά ορίζεται μέσω αντιδραστικών κανόνων (reactive rules) καθώς και εκείνων των οποίων οι ενέργειες εξαρτώνται από το περιβάλλον τους (context-aware systems) καθώς προσαρμόζουν τη συμπεριφορά τους ανάλογα με τις αλλαγές που παρατηρούν.

Στην πρώτη κατηγορία συστημάτων αρχικά προτείνεται μία τυπική σημασιο-

λογία των αναδραστικών κανόνων, βασισμένη στην άλγεβρα με κρυφούς τύπους, έπειτα μελετάται η τυπική ανάλυση και επαλήθευση των έξυπνων συστημάτων που ορίζονται μέσω αναδραστικών κανόνων με χρήση εξισωτικής λογικής, στη συνέχεια προτείνεται η χρήση λογικής βασισμένης στην αναγραφή όρων για την ανακάλυψη λαθών στη δομή των αναδραστικών κανόνων, όπως τερματισμού και σύγκλισης, και τέλος συγκρίνονται οι προτεινόμενες προσεγγίσεις.

Στη δεύτερη κατηγορία συστημάτων, τα οποία κατά κανόνα είναι και πιο σύνθετα, ορίζουμε την προδιαγραφή τους ως τη σύνθεση των προδιαγραφών των επιμέρους συστατικών τους, δηλ. των προδιαγραφών που περιγράφουν τους τύπους δεδομένων και τα συστατικά του συστήματος σαν Παρατηρήσιμα Συστήματα Μετάβασης - ένα είδος συμπεριφοριακού αντικειμένου - ώστε να μπορέσει να εκφραστεί κατάλληλα η αλληλεπίδραση των συστατικών των συστημάτων αυτων.

Στη συνέχεια με βάση τις προδιαγραφές και κάνοντας χρήση τεχνικών απόδειξης θεωρήματος είναι δυνατή η επαλήθευση ιδιοτήτων των συστημάτων. Το ότι ο δυναμικός χαρακτήρας του συστήματος προδιαγράφεται με κανόνες μετάβασης και χρησιμοποιώντας εξισωτική λογική ή λογική αναγραφής, κάνει τη μέθοδο αυτή καλύτερα κατανοητή και ευκολότερη από αντίστοιχες μεθόδους που προαπαιτούν βαθύτερη γνώση των τεχνικών απόδειξης θεωρήματος ή που βασιζονται σε λογικές ανώτερου επίπεδου.

Η εφαρμογή των προτεινόμενων μεθόδων παρουσιάζεται μέσα από τη μελέτη μιας σειράς περιπτώσεων που αντιστοιχούν σε σχετικά συστήματα και πρωτόκολλα. Τα ζητήματα ασφαλείας για ένα έξυπνο σύστημα είναι μεγάλης κρισιμότητας και γι' αυτό το λόγο δεν θα μπορούσαν να παραλειφθούν. Στα πλαίσια αυτά έχουν προδιαγραφεί και επαληθευθεί συστήματα έξυπνων πρακτόρων από τη διεθνή βιβλιογραφία.

Στο τελευταίο κομμάτι της διατριβής προτείνεται μια επέκταση της μεθοδολογίας μοντελοποίησης και απόδειξης ιδιοτήτων που υποστηρίζεται από τη γλώσσα CafeOBJ, κατάλληλη για την αυτοματοποίηση των αποδείξεων και τη χρήση πιο σύγχρονων εργαλειών ανάλυσης προδιαγραφών, με χρήση του συστήματος αυτόματης απόδειξης Athena. Προτείνεται μία μεθοδολογία αλληλεπίδρασης των δύο συστημάτων απόδειξης, ώστε να είμαστε σε θέση να εκμεταλλευτούμε τα πλεονεκτήματα του καθενός ξεχωριστά, και παρουσιάζονται παραδείγματα εφαρμογών της προτεινόμενης προσέγγισης σε πρωτόκολλα που μελετώνται συχνά για τη δοκιμή εργαλείων επαλήθευσης σύνθετων συστημάτων.

Λέξεις κλειδιά:
Έξυπνα Συστήματα, Τυπικές Μέθοδοι, Γλώσσες Αλγεβρικών Προδιαγραφών, Τυπική Επαλήθευση, CafeOBJ, Παρατηρήσιμα Συστήματα Μετάβασης, Απόδειξη Θεωρήματος, Επαγωγική Απόδειξη, Συμπεριφοριακές Προδιαγραφές, Άλγεβρα με Κρυμμένους Τύπους, Αφηρημένος Τύπος Δεδομένων, Αφηρημένη Μηχανή Καταστάσεων, Αντιδραστικοί Κανόνες, Συστήματα που Εξαρτώνται από το Περιβάλλον, Συστήματα Αυτόματης Απόδειξης, Athena.

Abstract

Formal methods are techniques, languages and tools based on mathematics, which provide an unambiguous, strict mathematical description or specification which is used for effective design, analysis and verification of desired properties of the system.

An important branch of formal methods are algebraic specification languages with a rigorous basis on mathematical logical systems or combinations of them. Such a language is CafeOBJ, an executable, new generation algebraic specification language, member of the OBJ family languages. Its main characteristic, that differentiates it from other formalisms, is its direct support to behavioural specification paradigm since it embeds special hidden sorts and behavioural operators in its syntax. Behavioural specification is based on hidden algebra and supports an object oriented style of algebraic specification. It also supports specification of distributed complex systems as abstract state machines and verification of safety properties of them through theorem proving techniques such as simultaneous induction and coinduction.

The scope of the thesis is the modelling and verification of intelligent agents using algebraic specification techniques.

By intelligent agents we refer to the systems used in the New or Semantic Web in order to achieve the need for reactivity, interaction and communication between its various components. An intelligent agent can be defined as an autonomous entity which observes through sensors and acts upon an environment using actuators (i.e. it is an agent) and directs its activity towards achieving goals.

Due to the special characteristics of such systems, and their increased development as well as their use in critical domains, the requirements for reliability, security and proper functionality has led to the use of formal methods for their analysis and the development of new methodologies oriented to these systems.

To this end, in this thesis an algebraic framework is proposed for the specification and verification of two basic types of intelligent systems, those whose behaviour is expressed in terms of reactive rules (reactive rule-based systems) and those who can sense the changes in their physical environment (context-aware systems), and adapt their behaviour accordingly.

In the first category of intelligent agents, we first give formal semantics, based on the hidden algebra formalism, to the basic reactive rule families, then we formally analyse and verify reactive rule-based systems using equational logic, next we propose the use of rewriting logic for the detection of structural errors of the rules, such as termination and confluence, and finally we compare the proposed approaches.

In the second type of agents, which are usually more complex, the specification of the system, is defined as the composition of the specifications of

its components, i.e. the specifications which describe data types as visible sorts and the various components of the system as Observational Transition Systems, a kind of behavioural object. In this way the complex interactions of such systems can be expressed in a natural and dynamic way.

Based on the specification, verification of properties of the intelligent system using theorem proving techniques is also feasible. The fact that the system is specified as a transition system, using equational or rewriting logic, makes the method easier to read, understand and learn than other related methods, which prerequisite deeper knowledge of theorem proving, or that are based in higher order logic for example.

To demonstrate the applicability and effectiveness of the proposed methodologies, a number of case studies are conducted. Security aspects of intelligent systems are of major importance, and it was inevitable to take them into account. To this end we present the verification of the behaviour of various intelligent systems from the literature.

In the last section of the thesis, an extension of the specification and verification methodology, supported by the CafeOBJ language, is proposed that provides more automation for the proofs and allows the use of more conventional verification tools, by integrating it with the Athena automated theorem proving system. The proposed methodology is based on the interaction of the two languages so as to be able to exploit the nice properties of each method and thus have better results in the verification process. Finally, a number of applications of the methodology in protocols that are often used as case studies, especially for tools dedicated to the verification of complex systems, are presented.

Keywords:
Intelligent agents, Formal Methods, Algebraic Specification Languages, Formal Verification, CafeOBJ, Observational Transition Systems, Theorem Proving, Induction, Behavioural Specifications, Hidden Algebra, Abstract Data Type, Abstract State Machine, Reactive rules, Context-aware systems, Automated theorem provers, Athena.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά τον κ. Στεφανέα για τη συνεχή υποστήριξη και την καθοδήγηση που μου παρείχε όλα αυτά τα χρόνια καθώς και για την εμπιστοσύνη που μου έδειξε αλλά και την ενθάρυνση να ανακαλύψω και να ακολουθήσω τα ερευνητικά μου ενδιαφέροντα.

Ακόμα χρωστάω ένα μεγάλο ευχαριστώ στον κ. Φράγκο για όλη την στήριξη που μου παρείχε με κάθε μέσο που διέθετε όπως επίσης και στην κα Λαμπροπούλου για όλα όσα έχει κάνει για εμένα.

Θα ήθελα να ευχαριστήσω και τον κ. Αρκούδα για τις ερευνητικές συζητήσεις που είχαμε και τα χρήσιμα σχόλια που μου έδωσε.

Τέλος θα ήθελα να ευχαριστήσω την οικογένεια μου για όσα μου προσέφερε και ιδιαίτερα τον Νίκο Τριανταφύλλου για όλα.

Acknowledgments

This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: THALIS.

Contents

1 Introduction

Formal methods are mathematically based techniques for the specification, verification and development of reliable software and hardware systems. Formal methods provide a wide range of techniques for reasoning about computer systems that can contribute to the reliability and robustness of their design. Formal design can be seen as a three step process [1]:

1. Formal Specification: During the formal specification phase, the engineer rigorously defines a system using a formal language. Such languages are fixed grammars which allow users to model complex structures out of predefined types. This process of formal specification is helpful in understanding the requirements of a system.

2. Verification: The process of formal verification can prove that a system does not have defects or that it does satisfy desirable properties, which is very important for software systems. Critical system requirements like safety and liveness properties play important role in the system specification, development and testing.

3. Implementation: Once the model has been specified and verified, it is implemented by converting the specification into code.

Algebraic specifications, one of the major formal methods, use algebraic modeling for the specification of the systems and then the designs are verified against requirements using algebraic techniques. Developments of algebraic specification show that evolution of systems can be neatly modeled by rewriting logic algebraically. It also shows that behavior of systems can also be nicely modeled by hidden algebras [3]. CafeOBJ [2] is an algebraic specification language and processor and it is used for the formal analysis and verification of software systems. CafeOBJ adopts hidden algebra and rewriting logic as its underlying logics. Its application areas include design and validation of complex systems such as component based systems, security systems and protocols, computer language systems, and many more.

On the other hand, nowadays, there is a strong demand for computer systems to become more pervasive, to communicate and interact with each other and to be able to handle reactivity on the Web efficiently. These advances in computer technology have resulted in intelligent computer systems, which are often hard to understand and to program. Also, as the design of computer software and hardware becomes more complex and ubiquitous systems are developed and used in critical domains, the need for such systems to behave correctly increases.

The Semantic Web - also called New or Dynamic Web - is an extension of the current web in which information is given well defined meaning, better enabling computers and people to work in cooperation [4]. In order to

achieve reactivity and communication among the different components, the New Web employs intelligent agents. An intelligent agent (IA) can be defined as "a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future" [5].

A usual way to program intelligent agents and to define their behavior is through the use of reactive rules. More precisely, reactive rules are used for programming rule-based, reactive systems, which have the ability to detect events and respond to them automatically in a timely manner. This creates a main category of intelligent agents, the reactive rule-based systems. Such systems are needed on the Web for bridging the gap between the existing Web, where data sources can only be accessed to obtain information, and the dynamic Web, where data sources are enriched with reactive behavior. One more complex type of intelligent agents is the so called context-aware or adaptive systems. Context-aware computing refers to a general class of mobile systems that can sense the changes in their physical environment, and adapt their behavior accordingly. Context-aware systems are a component of a ubiquitous computing or pervasive computing environment. The common characteristic of the above systems, and of most intelligent agents, is that they can present complex and unpredictable behavior, and thus there is a strong need for their formal analysis. We believe that algebraic specifications are a suitable approach for specifying the dynamic behavior of these agents, as they provide an intuitive approach to model complex distributed systems. However, sometimes it is useful to use more conventional theorem proving systems to automate the verification process and to ensure the soundness of the proofs, especially when the systems become more critical.

To this end, in this thesis we study the unpredictable behavior of intelligent agents used in the Semantic Web, and we propose appropriate methodologies, by exploiting the capabilities of the CafeOBJ language, for agent reasoning and verification. The proposed approaches are validated by applying them in illustrating case studies. We then investigate further potentials of the CafeOBJ verification methodology, by combining it with the Athena automated theorem prover. In this way, critical and complex intelligent systems can be specified using traditional algebraic methods that offer nice ways to model software systems, such as CafeOBJ, and then using the combination of CafeOBJ and Athena proof systems, they can be verified in a more sound and automatic way.

The rest of the thesis is organized as follows: in chapter 3 we formally analyze and verify reactive rule-based systems. In chapter 4 we propose a methodology for the verification of context-aware adaptive systems. In chapter 5, we propose a methodology that integrates the CafeOBJ language with the Athena automated theorem prover.

2 Preliminaries - Theoretical Background

Formal methods use mathematics and logics to formalize requirements, designs and programs of systems, and verify that there are no inconsistencies in requirements, designs enjoy requirements, and programs conforms to designs [6]. Algebraic specification techniques have been developed in formal methods and several algebraic specification languages and processors have been proposed. CafeOBJ [7] is one such language and processor. Behavioral specifications [8, 9] are algebraic specifications of systems behavior and can be described in CafeOBJ.

2.1 Algebraic specifications

In more details, algebraic specification is a software engineering technique for formally specifying systems behavior, which aims at:

- formally defining types of data, and mathematical operations on those data types,
- abstracting implementation details, such as the size of representations and the efficiency of obtaining outcome of computations,
- formalizing the computations and operations on data types,
- allowing for automation by formally restricting operations to this limited set of behaviors and data types.

An algebraic specification achieves these goals by defining one or more data types, and specifying a collection of functions that operate on those data types. These functions can be divided into two classes:

- constructor functions: functions that create or initialize the data elements, or construct complex elements from simpler ones, and
- additional functions: functions that operate on the data types, and are defined in terms of the constructor functions.

2.2 Behavioral specification

Behavioral specification [8–10] provides a novel generalization of ordinary algebraic specification. It characterizes how objects (and systems) behave, not how they are implemented. This new form of abstraction can be very powerful in the specification and verification of software systems since it naturally embeds other useful paradigms such as concurrency, object-orientation, constraints, nondeterminism, etc. (see [8] for details). Behavioral abstraction is achieved by using specification with hidden sorts and a behavioral concept

of satisfaction based on the idea of in distinguishability of states that are observationally the same, which also generalizes process algebra and transition systems (see [8]).

2.3 Hidden Algebra

Hidden Algebra is an approach for giving semantics to concurrent distributed object oriented systems, as well as to software systems in general. It is a version of behavioral type in the object oriented paradigm [11], called behavioral specification.

For a set of sorts S, we say that the set A is S-sorted if it can be regarded as a family of sub-sets as $A = \cup\{A_s\}_{s \in S}$. Using this set of sorts, we can define a signature Σ as the pair $<S, F>$ where F is a set of function symbols. Such that F is equipped with a mapping $F \rightarrow S^* \times S$ meaning that each $f \in F$, $f : s_1 \times \cdots \times s_n \rightarrow s$. Then the type (or rank) of f is defined as $rank(f) = s_1 \ldots s_n s \in S^*$. Given a signature as above, a Σ-Algebra A consists of a non-empty family of carrier sets $\{A_s\}_{s \in S}$ and a total function $f^A : A_{s_1} \times \ldots \times A_{s_n} \rightarrow A_s$ for each function symbol $f : s_1 \times \cdots \times s_n \rightarrow s \in F$.

A Σ-homomorphism between two Σ-algebras A and B, denoted by $h : A \rightarrow B$, is a family of maps $\{h_s : A_s \rightarrow B_s\}_{s \in S}$ that preserves the operators. By imposing a partial ordering on the sorts we get an Order Sorted Σ-Algebra (OSA).

An *order sorted signature* is a triple $(S, \leq, \Sigma)$ such that (S, Σ) is a many-sorted signature, $(S, \leq)$ is a poset, and the operators satisfy the following monotonicity condition; $\Sigma \in \Sigma_{w1,s1} \cap \Sigma_{w2,s2}$ and $w1 \leq w2$ implies $s1 \leq s2$. Given a many-sorted signature, an (S, Σ)-algebra A is a family of sets $\{A_s \mid s \in S\}$ called the carriers of A, together with a function $A_\Sigma :_w \rightarrow A_s$ for each Σ in $\Sigma_{w,s}$. Where $A_w = A_{s1} \times \cdots \times A_{sn}$ and $w = s_1 \ldots s_n$.

Let $(S, \leq, \Sigma)$ be an order sorted signature. A $(S, \leq, \Sigma)$-*algebra* is a (S, Σ)-algebra A such that, $s \leq s'$ in S implies $A_s \subseteq A_{s'}$ and $\Sigma \in \Sigma_{w1,s1} \cap \Sigma_{w2,s2}$ and $w1 \leq w2$ implies $A_\Sigma :_{w1} \rightarrow A_{s1}$ equals $A_\Sigma :_{w2} \rightarrow A_{s2}$ *on* A_{w1} [8]. The purpose of the formalization of order sorted signatures is to define sorts (similar to classes in OO), functions (similar to methods in OO) and inheritance between the sorts.

In Hidden Algebra [8] two kinds of sorts exist: visible sorts and hidden sorts. Visible sorts represent the data part of a specification while hidden sorts denote the state of an abstract machine.

Given a signature $(S, \leq, \Sigma)$ and a subset $H \subset S$ denoting the hidden sorts, a hidden algebra (or a hidden model in general) A interprets the visible sorts V and the operations Ψ of the visible sorts as a fixed model D (the data model, say an order sorted algebra) such that $A\restriction_{V,\Psi} = D$ (where $\restriction$ is the model reduct). Given two signatures $(S, \leq, \Sigma)$ and $(S', \leq, \Sigma')$ a signature morphism $\phi : (S, \leq, \Sigma) \rightarrow (S', \leq, \Sigma')$ consists of a mapping z on

sorts that preserves the partial ordering, i.e. for $s \leq s'$ then $z(s) \leq z(s')$ and an indexed mapping on operators g, such that $\{g_{s_1 \ldots s_n} : \Sigma_{s_1 \ldots s_n s} \rightarrow \Sigma'_{z(s_1) \ldots z(s_n) f(s)}\}_{s_1 \ldots s_n s \in S, n \geq 0}$.

We will refer to operators whose arguments contain a hidden sort and/or whose arguments returned value is a hidden sort, as hidden or behavioral operators.

The hidden signature morphism $\phi : (S, H, \leq, \Sigma) \rightarrow (S', H', \leq, \Sigma')$ preserves the visible and hidden part of the signatures and obeys to the following conditions: (a) g maps each behavioral operator to a behavioral operator, (b) if $z(h) < z(h')$ for arbitrary visible sorts h and h' then $h < h'$ and (c) if $\sigma' \in \Sigma_{w's'}$ is a behavioral operator where $w \in (S \bigcup H)^*$ and some sort of w is hidden, then $\sigma' = g(\sigma)$ for some behavioral operator $\sigma \in \Sigma$.

2.4 CafeOBJ

2.4.1 The behavioral specification paradigm

CafeOBJ behavioral specification paradigm is based on coherent hidden algebra (abbreviated CHA) of [9], which is both a simplification and extension of classical hidden algebra of [8] in several directions, most notably by allowing operations with multiple hidden sorts in the arity. Coherent hidden algebra comes very close to the observational logic of Bidoit and Hennicker [10]. CafeOBJ directly supports behavioral specification and its proof theory through special language constructs, such as [12]:

- hidden sorts (for states of systems),
- behavioral operations (for direct actions and observations on states of systems),
- behavioral coherence declarations for (non-behavioral) operations (which may be either derived (indirect) observations or constructors on states of systems), and
- behavioral axioms (stating behavioral satisfaction).

The advanced coinduction proof method receives support in CafeOBJ via a default (candidate) coinduction relation (denoted as =*=). Coinduction can be used either in the classical hidden algebra sense [8] for proving behavioral equivalence of states of objects, or for proving behavioral transitions (which appear when applying behavioral abstraction to rewriting logic). Besides language constructs, CafeOBJ supports behavioral specification and verification by several methodologies. It currently highlights a methodology for concurrent object composition which features high reusability not only of specification code but also of verifications [7, 13]. Behavioral specification in CafeOBJ may also be effectively used as an object-oriented

(state-oriented) alternative for classical data-oriented specifications. Experiments seem to indicate that an object-oriented style of specification even of basic data types (such as sets, lists, etc.) may lead to higher simplicity of code and drastic simplification of verification process [7].

Behavioral specification is reflected at the execution level by the concept of behavioral rewriting [7,9] which refines ordinary rewriting with a condition ensuring the correctness of the use of behavioral equations in proving strict equalities.

2.4.2 Behavioral Object Composition

CafeOBJ supports the object composition [14–16] of distributed static and dynamic systems that allows both reusability of specification code and proofs. Engineers can start from small valid and easy to handle specifications and incrementally combine them to build the complete specification of the whole system. Reusing specifications is done by projection operators. Projection operators are defined for the composing objects in order to obtain their states from the state of composed object. Using them, all methods of the composed object are related to those of the composing objects.

Hidden Algebra extends ordinary many sorted algebra [17] with extra sorts representing the "states" of an object or an abstract machine or a behavioral object (BO). Behavioral Object Composition has been defined formally in [15]. The objects with no components are called base-level objects. A composition is represented with arrows whose heads are diamonds, and if necessary, qualified by the numbers of components (1 for one and * for many). Also, the circle at the tail of an arrow denotes that the composite object contains an arbitrary amount of components. We can retrieve the state of the component objects via Projection Operators [13,16], special observers of the composite object that given a state of the composite object return the state of one base-level object. There are several ways to compose an object. Parallel Composition with Synchronization occurs when the changes in the state of one object may alter the state of an object in the same level. In respect to the number of objects that compose a composite object, we have Dynamic Composition if the number of component objects is not fixed. Else the composition is called Static. Hierarchical behavioral object composition can be represented in UML [18] notation in the figure below.

2.4.3 Basic syntax and notation

CafeOBJ algebraic specification language [7] can be used for the specification and verification of complex software systems. The basic units of CafeOBJ are its modules. There are two kinds of modules in CafeOBJ, tight and loose modules. A tight module only accepts the smallest implementation

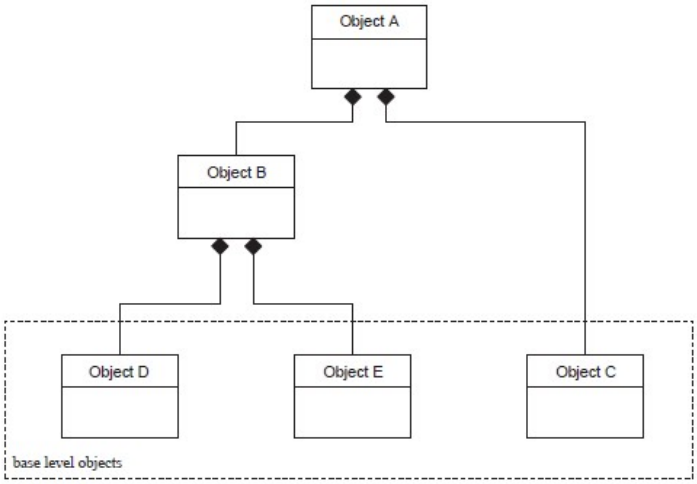


Figure 1: UML notation for hierarchical behavioral object composition.

that satisfies what are specified in the module, while a loose module can accept any implementations (that satisfy them). A tight module is declared with the keyword *mod*!, and a loose module with the keyword *mod*∗.

In CafeOBJ modules, we can declare module imports, sorts, operators, variables and equations. Operators without arguments are called constants. Built-in operators denoting logical connectives can be used to declare negation, conjunction, disjunction, implication, and exclusive disjunction. Operators can have attributes such as *comm* that specifies that the binary operator is commutative. Conditional equations can also be declared inside a module. Operators are declared with the keyword *op* (or *ops* if there are many). The constructor operators of the sorts are declared with the attribute *constr*. The non-constructor operators, or some properties of the operators are defined in equations.

Here for example, is the specification of the list data structure in CafeOBJ. A line that starts with `--`, `-->`, `**`, or `**>` is a comment.

```
--> trivial set of elements
mod* TRIV* {[Elt]}
--> parameterized list
mod* LIST (X :: TRIV*) {
[Nil NnList < List]
op nil : -> Nil {constr}
op _|_ : Elt List -> NnList {constr}
-- equality on the sort List
op _=_ : List List -> Bool {comm}
eq (L:List = L) = true .
ceq L1:List = L2:List if (L1 = L2) .
}
```

As we can see, the module `TRIV*` only declares the sort `Elt`. The module

`LIST` defines a parameterized list structure. `[Nil NnList < List]` declares three sorts and sub-sort relations among them; sorts `Nil` and `NnList` are sub-sorts (i.e. are interpreted as subsets) of sort `List`. The operators `nil` and `|` are defined with constructor attribute and thus define the way the sorts `Nil`, `NnList`, and `List` are constructed. For more details we refer the interested reader to [19].

2.4.4 Equational and rewriting logic

CafeOBJ supports both equational theory and rewrite theory specifications. State transitions are described in equations in the former and in rewriting rules in the latter. Equational theory specification is used for interactive theorem proving whereas for rewrite theory specification, CafeOBJ can conduct exhaustive searches. In [20] an attempt to combine the above is presented. They describe a way to theorem prove that rewrite theory specifications have invariant properties by proof score writing.

In equational logic the *transitions* between the states of the system are modeled with constructor operators that change its state (which is denoted by a sort, say *State*). The structure of a state is abstracted by the observation operators (or *observers*), each one returning an observable information about the state. The meaning of an observer is formally described by means of (conditional) equations, depending on whether the transition has an effective condition or not (a condition that must hold for the transition to be applied). These equations define how the value of each observer changes after the application of a transition rule.

Rewriting logic in CafeOBJ is based on a simplified version of Meseguer's rewriting logic [21] for concurrent systems which gives an extension of traditional algebraic specification towards concurrency. CafeOBJ design does not fully support labeled rewriting logic which permits full reasoning about multiple transitions between states, but supports reasoning about the existence of transitions between states (or configurations) of concurrent systems via a built-in predicate (denoted as `==>`) with dynamic definition encoding both the proof theory of rewriting logic and the user defined transitions [22]. This predicate evaluates to true whenever there exists a transition from the left hand side argument to the right hand side argument [22]. More precisely, for a ground term t, a pattern p and an optional condition c, CafeOBJ can traverse all the terms reachable from t wrt. transitions in a breadth-first manner and find terms (called solutions) such that they are matched with p and c holds for them. This can be done using the command: `red t =(k,d)=>* p [suchThat c]`, where k is the maximum number of solutions and d is the maximum depth of search. Also, a natural number, id is assigned to each term visited by a search and then by using the command `show path id` a transition path to the term identified by id is displayed. Typically, the command is used to display a transition path to a solution

found by a search from t [20].

2.5 Observational Transition Systems

An Observational Transition System (OTS) is a transition system written in terms of equations [23, 24] and is a proper subclass of behavioral specifications [8]. We assume that there exists a universal state space Y and that each data type we need to use has been declared in advance. An OTS S is defined as the triplet $S = \langle O, I, T \rangle$ where [25]:

- O is a set of observers. Each $o \in O$ is a function $o : Y\, D_{o1} \dots D_{om} \rightarrow Do$, where D_i denotes some data-type. Given an OTS S and two states u_1, u_2 the equivalence between them is defined with respect to the values returned by the observers, i.e. $u_1 =_S u_2$ if and only if for each $o \in O$, $o(u_1, x_1, \dots, x_m) = o(u_2, x_1, \dots, x_m)$ for all $x_1 \in Do_1, \dots, x_m \in Do_m$.
- I is the set of initial states, such that $I \subseteq Y$.
- T is a set of conditional transitions. Each $t \in T$ is a function $t : Y\, D_{t1} \dots D_{tn} \rightarrow Y$. Each transition t, together with any other parameters $y_1, \dots, y_n$, preserves the equivalence between two states, i.e. if $u_1 =_S u_2$, then for each $t \in T$, $t(u_1, y_1, \dots, y_n) =_S t(u_2, y_1, \dots, y_n)$ for all $y_1 \in D_{t1}, \dots, y_n \in D_{tn}$. Each t has the effective condition $c\text{-}t : Y\, D_{t1} \dots D_{tn} \rightarrow Bool$. If $\neg\ c\text{-}t(u, y_1, \dots, y_n)$, then $t(u, y_1, \dots, y_n) =_S u$. $t(u, y_1, \dots, y_n)$ is called a successor state of a state u. We write $u \rightsquigarrow_S u'$ iff a state $u' \in Y$ is a successor state of a state $u \in Y$.

An *execution* of an OTS S is an infinite sequence $u_0, u_1, \dots$ of states satisfying the following:

- *Initiation*: $u_0 \in I$,
- *Consecution*: For each $i \in N$, there exists $t \in T$ such that $u_i + 1 =_S t(u_i)$.

Assume that ϵ_s is the set of all executions obtained from S. A state $u \in Y$ appears in an execution $u_0, u_1, \dots$ of an OTS S, denoted by $u \in u_0, u_1, \dots$ if there exists $i \in N$ such that $u =_S u_i$.

A state $u \in Y$ is called reachable with respect to an OTS S, if and only if there exists an execution $e \in \epsilon_s$ such that $u \in e$. Let R_S be the set of all reachable states with respect to S[1].

[1] R_S is the type denoting the set of all reachable states wrt S. Also Sys denotes R_S but not Y if the constructor-based logic is adopted, which is the current logic underlying the OTS/CafeOBJ method.

CafeOBJ is used to specify Observational Transition Systems. The universal state space Y of an OTS is denoted in CafeOBJ by a hidden sort, say Sys. Each observer is denoted by an observation operator. Any state in I is denoted by a constant and each transition by an action operator. The transitions are defined by describing what the value, returned by each observer in the successor state, becomes when the transitions are applied in a state u. Finally, when the effective condition holds, this is expressed by a conditional equation.

2.6 Timed Observational Transition Systems

Timed observational transition systems (TOTSs) are OTSs that are evolved by introducing clock observers in order to deal with timing. Again Y denotes a universal state space. Let $\mathbb{B}$, $\mathbb{N}$ and $\mathbb{R}^+$ be a set of truth values, a set of natural numbers and a set of non-negative real numbers, respectively. A TOTS $S = \langle O, I, T \cup \{tick_r \mid r \in \mathbb{R}^+\}\rangle$ where [9]:

1. O is a set of observers. The set $O = D \cup C$ is classified into the set D of discrete observers and the set C of clock observers. Clock observers may be also called clocks. The discrete observers are defined in the same way as the observers of an ordinary OTS.

2. I is the set of initial states such that $I \subseteq Y$.

3. $T \cup \{tick_r \mid r \in \mathbb{R}^+\}$ is a set of conditional transitions. Each $t \in T \cup \{tick_r \mid r \in \mathbb{R}^+\}$ is a function $t : Y \to Y$.

For each clock observer $o \in C$ where $o : Y \to D$, D is a subset (subtype) of $\mathbb{R}^+ \cup \{\infty\}$. For each $t \in T$, there are two clocks $l_t : Y \to \mathbb{R}^+$ and $u_t : Y \to \{\mathbb{R}^+\} \cup \{\infty\}$, which return the lower and upper bounds of t, respectively. They are basically used to force t to be executed, or applied between the lower bound returned by l_t and the upper bound returned by u_t. There is also one special clock, called $now : Y \to \mathbb{R}^+$. It serves as the master clock and returns the time amount that has passed after starting the execution of S. now initially returns $zero$. Thus, C contains the two clocks l_t and u_t for each $t \in T$, and the master clock now. For each $t \in T$, its effective condition consists of the timing part and the non-timing part. The non-timing part is denoted by c_t. Given a state $u \in Y$, the timing effective condition is $l_t \leq now(u)$ .Each $tick_r$ is a time advancing transition. Given a state $u \in Y$, for each $tick_r$, its effective condition is $now(u) + r \leq u_t(u)$ for each $t \in T$, and $now(tick_r(u))$ is $now(u) + r$ if the effective condition of $tick_r$ is true in u.

2.7 The OTS/CafeOBJ methodology

One of the main advantages of using algebraic specifications is the ability to verify that the specified system preserves critical properties. For this

reason, CafeOBJ is equipped with its processor called the CafeOBJ system, which serves as a theorem prover. The CafeOBJ system verifies the desired properties by using the equations of the theory that defines the OTS as left to right rewrite rules. The theorem proving technique that is used in order to verify the desired properties is called the OTS/CafeOBJ method or Proof Scores approach [24, 26] and is a computer human interactive method. A Proof Score is a plan to verify that a property holds for a specification. This is implemented as a set of instructions written by a human to the proof engine, such that when executed, and if everything evaluates as expected, a desired theorem is proved (computer human interaction).

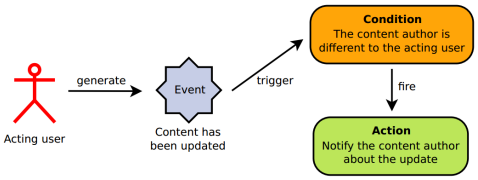
This method hides the tedious calculations done by the machine, and reveals the proof plan created by the human. The main differences from model checking are the ability to deal with systems that have an infinite number of states, the natural affinity for abstraction and finally the emphasis on re-usability [26]. Also, this approach to verification is sometimes preferable to fully automated ones, because the latter often fails to convey to the user an understanding of the proof. Automated theorem provers can reason if a property holds or not, but usually do not provide enough feedback when the proof fails. So the user is unaware whether the failure is due to some design error, if additional input is required or if finally a lemma is needed. In contrast, in the OTS/CafeOBJ method if the proof fails, either an expression or a false message is returned to the user. In both cases the system's states that returned these outputs are fully defined and the user can intervene and either provide additional input (first case) or reason and formally prove that the case should not be reached by the system (second case) or even refine the specification because the property does not hold (second case). Another benefit of this method, is that the proof plan is executed over the specification. As a result the level of abstraction of the proof equals the level of the specification. Usually with this method the goal is to verify design not implementation. A more thorough analysis of the benefits of the OTS/CafeOBJ verification methodology as well as of its disadvantages will be presented in the last chapter of the thesis. In the same chapter a more detailed comparison of this method with other proving systems and verification techniques will be given as well.

The OTS/CafeOBJ method can be used to verify both liveness properties (something will eventually happen) and invariants (something holds) of the specified system. In this thesis, we focus on the second type of properties. A property for a system is called invariant if it holds in any given reachable state of the system. Roughly speaking, an invariant property can be proved with the Proof Score method in 4 steps. First, we formally express the property we want to prove as a predicate in CafeOBJ terms in a module. Next we must write the inductive step as a predicate that contains two states s and s', denoting an arbitrary state and its successor respectively, and that defines that if the invariant holds in s then that implies that it holds in s'.

Then we can ask CafeOBJ to prove via term rewriting (using the reduce command), if the property holds for an arbitrary initial state. Finally, using all the transition rules in turn, we must instantiate s' and ask CafeOBJ to prove the inductive step for each case.

In a nutshell, after asking CafeOBJ to prove such an expression three results might be returned by the system. If true is returned this means that the proof was successful. If a CafeOBJ term is returned, different to true or false, this means that there exist some terms that the system cannot fully reduce. The user must split the case, by stating that the problematic term equals to true and false in turn (computer-human interactive method). This creates two new proof obligations. This is known as case splitting. A usual example is the effective conditions of transitions in arbitrary sates. Finally, if false is returned, two things might hold. The property does not hold for our system, or the case that returned false is unreachable. In the first case we have found a counter example case and it might be necessary to redesign our system. In the second case we must create a lemma, a new invariant property, which states that the case that returned false is not reachable. Of course in this case the lemma must be proved separately. The OTS/CafeOBJ methodology as well as its steps and possible outputs will be presented in details and explained through the running examples and case studies of the thesis.

3 Reactive Rules



The Web has traditionally been perceived as a distributed repository of hypermedia documents and data sources with clients (in general browsers) that retrieve documents and data, and servers that store them. Although reflecting a widespread use of the Web, this perception is not accurate. With the emergence of Web applications, Web Services, and Web 2.0, the Web has become much more dynamic.

Reactivity on the Web, i.e. the ability to detect events and respond to them automatically in a timely manner, is needed for bridging the gap between the passive Web, where data sources can only be accessed to obtain information, and the dynamic Web, where data sources are enriched with reactive behavior.

Reactivity is a broad notion that spans Web applications such as e-commerce platforms that react to user input (e.g. putting an item into the shopping basket), Web Services that react to notifications or service requests (e.g. SOAP messages), and distributed Web information systems that react to updates in other systems elsewhere on the Web (e.g. update propagation among biological Web databases). Such Web nodes (applications, sites, services, agents, etc.) constantly react to events bringing new information or making existing information outdated and change the content of data sources. Programming reactive behavior entails (1) detecting situations that require a reaction and (2) responding with an appropriate state-changing action. We present in the following some applications of reactive behavior on the Web [27].

Shopping Cart. This example shows how a simple reactive rule set calculates the shopping discount of a customer. The business rules describing the discount allocation policy is listed hereafter:

1. If the total amount of the customer's shopping is higher than 100, then perform a discount of 10%.

2. If it is the first shopping of the customer, then perform a discount of 5%.

3. If the client has a gold status and buys more than 5 discounted items, then perform an additional discount of 2%.

Rules 1 and 2 must not be applied for the same customer, the first rule has the priority against the second. The third rule is applied only if rule 1 or rule 2 have been applied.

Those policies might be taken into account by a reactive rule service (Web service, procedural application, etc.). This service receives the customer and his shopping cart information as input. The discount calculation is then processed following the previous rules and returns the discount value to the service caller.

Credit Analysis. This example shows how a simple reactive rule set defines a loan acceptance service. It determines whether a loan is accepted, depending on the client's history and the loan request duration. A client's score is calculated according to the following business policy. If the client's score is high enough, the loan is accepted and its rate is calculated.

1. If the loan duration is lower than five years then set the loan rate to 4% and add 5 to the score, else set it to 6%.
2. If the client has filed a bankruptcy, subtract 5 to the score.
3. If the client's salary is between 20000 and 40000, add 10 to the score.
4. If the client's salary is greater than 40000, add 15 to the score.
5. If the score is upper than 15, then the loan is accepted.

Those policies are usually implemented by a rule service (Web service, application). This service receives the loan request as input information, applies the rule on them in order to check the acceptance, and finally returns to the caller the loan characteristics.

Distributed Information Portal. Many data sources on the Web are evolving in the sense that they change their content over time in reaction to events bringing new information or making existing information outdated. Often, such changes must be mirrored in data on other Web nodes updates need to be propagated. For Web applications, such as distributed information portals, where data is distributed over the Web and part of it is replicated, update propagation is a prerequisite for keeping data consistent. As a concrete application example, consider the setting of several distributed Web sites of a fictitious scientific community of historians called the Eighteenth Century Studies Society (ECSS). ECSS is subdivided into participating universities, thematic working groups, and project management. Universities, working groups, and project management have each their own Web site, which is maintained and administered locally. The different Web sites are autonomous, but cooperate to evolve together and mirror relevant changes from other Web sites. The ECSS Web sites maintain

(XML or RDF) data about members, publications, meetings, library books, and newsletters. Data is often shared, for example a member's personal data is present at his home university, at the management node, and in the working groups he participates in. Such shared data needs to be kept consistent among different nodes. This can be realized by communicating changes as events between the different nodes using reactive rules. Events that occur in this community include changes in the personal data of members, keeping track of the inventory of the community-owned library, or simply announcing information from email newsletters to interested working groups. These events require reactions such as updates, deletion, alteration, or propagation of data, which can also be implemented using reactive rules. Full member management of the ECSS community, a community-owned and distributed virtual library (e.g., lending books, monitions, reservations), meeting organization (e.g., scheduling panel moderators), and newsletter distribution are desirable features of such a Web-based information portal. And all these can be elegantly implemented by means of reactive rules.

Many Web-based systems need to have the capability to react not only to simple events but also to situations represented by temporal combinations of events. For communicating events on the Web two strategies are possible: the push strategy, i.e. a Web node informs (possibly) interested Web nodes about events, and the pull strategy, i.e. interested Web nodes query periodically (poll) persistent data found at other Web nodes in order to determine changes. Both strategies are useful. A push strategy has several advantages over a strategy of periodical polling: it allows faster reaction, avoids unnecessary network traffic, and saves local resources.

Different approaches can be followed for implementing Web applications having the capabilities mentioned above. Compared with general purpose programming languages and frameworks, rule-based programming brings in declarativity, fine-grain modularity, and higher abstraction. Moreover, modern rule-based frameworks add natural-language-like syntax and support for the life cycle of rules. All these features make it easier to write, understand, and maintain rule-based applications, including for non-technical users.

Reactive rules are thus high-level, elegant means to implement reactive Web applications whose architecture imply more than one Web components/nodes and their communication is based on exchanging events. The issues of using, developing and analyzing reactive rules begin to play an increasingly important role within business strategy on the Web and event-driven applications are being more widely deployed.

In this chapter we first present formal semantics for some of the most common families of Reactive rules, based on the formalism of Hidden Algebra. We propose the use of the CafeOBJ specification language, which adopts hidden algebra as its underlying logic, for the verification of safety properties for reactive rule-based intelligent agents. Next, we propose a methodology, based on rewriting logic specifications written in CafeOBJ,

for verifying safety properties about systems whose behavior is expressed in terms of reactive rules but also for reasoning about structural errors of the rules. Finally, we revise the proposed verification methodologies, and we present a formal framework which allows reasoning about the specified rule-based system, verification of safety properties of the rules, detection of termination and confluence errors of the rules, and provides a clear understanding of the rule based system by simulating the execution behavior of the rules.

3.1 On the Algebraic Semantics of Reactive Rules

We present a Hidden Algebra approach that can be used to formally define some of the most common families of Reactive Rules. While there are many approaches in terms of expressing the meaning of reactive rules, our contribution focuses on the foundations for creating libraries of rules with verified behavior and allows the composition of these rules in order to create more complex rule bases that preserve desired properties. This can be achieved due to the strong modularisation properties of hidden algebra, such as information hiding, renaming and sum. Hidden algebra has been successfully applied to system's design verification [8]. By expressing reactive rules in the same framework we could reason not only about the rules but also on their behavior in particular systems.

In more details, in the following we give Observational Transition System (OTS) semantics for production (PR), event condition action (ECA) and knowledge representation rules (KR) as well as for complex event processing (CEP), and present some case studies. This semantics will allow the mapping between Rule Markup Languages and Behavioral Algebraic Specification Languages. Verification techniques for reactive rules, will provide automated reasoning capabilities and support the development of new rule based policies and trust models.

Reactive Rules for event driven applications started to be used extensively during the 1990s. The interest was on rules that specify the behavior of systems that trigger actions as response to the detection of events from the environment. As the authors of [28] point out, today's research on Event Driven Architecture IT infrastructures like on-Demand or Utility Computing, Real- Time Enterprise, Business Activity Management and so on, is intensive. In addition, a new push has been given in the field due to the strong demand of the web community for event processing functionalities for Semantic Web Markup Rule Languages such as RuleML.

Many different approaches to reactive event processing have been developed over the years. As analyzed in [28], in active databases the focus is on the support of automatic triggering of global rules in response to either internal updates or external events. In event notification and messaging systems, the focus is on the sequence of events in a given context, while in event/action logics it is on the inferences that can be made from the fact that certain events have occurred. For a survey on the event/action/state processing space we refer the interested reader to [29].

The most important families of reactive rules used in the Semantic Web community are Production rules, Event Condition Action (ECA) rules and Knowledge Representation (KR) rules. Also great interest is shown for the definition of Complex Events. Here we will present an Observational Transition System (OTS) semantics for the rule families depicted in the figure below.

3.1.1 Production Rules and OTS semantics

Production rule systems use a set of condition-action rules cyclically invoking (assert, retract, etc.) actions when tests over their working memory succeed. Their syntax is: $if\ C_i\ do\ A_i$, where A_i denotes an action that must be applied automatically by the system when it detects that the conditions denoted as C_i hold.

Our aim is to provide an OTS semantics to production rules. The semantics of these actions A_i could be those of transition rules in an OTS, because both transition rules and actions A_i cause explicit changes to the state of the system when they are applied. The conditions defined above can be mapped to (or be part of) the effective conditions of the transition rules.

So at first, it looks like the OTS semantics of production rules are quite straightforward. But a closer look will reveal that there is an error in the previous reasoning. In the OTS approach we can define under which conditions the transitions will be successful but it is not possible to enforce the application of those transitions on a state. On the other hand the production rules system must react in the desired way when the conditions are met. So the semantics of production rules in OTS should be the enforcement of the application of a transition rule in a system state.

Here, we will attempt to give such semantics by providing a typing on the states of an OTS. Then we will define an OTS that contains only transition rules from the typed states, which can be regarded as a sort of typed OTS. Recall that an OTS corresponds to an order sorted hidden algebra $(S, \leq, \Sigma)$. We will use the hidden sorts, defined as $H \subset S$, to produce the typing system. For the following we assume a set of hidden sorts H, denoting the state space of the OTS Y, meaning that $\forall u \in Y$, $\exists\, h \in H$ such that the sort of u belongs to the sort h.

Definition 1. Suppose that the OTS contains n-transition rules: $\tau_i : H\ D_{1_i}\ ...\ D_{n_i} \rightarrow H$, $i \in (1, \ldots, N)$ and that each transition rule is affected by the effective condition c-τ_i : $HD_{1_i} \ldots D_{n_i} \rightarrow Bool$. For each transition τ_i and

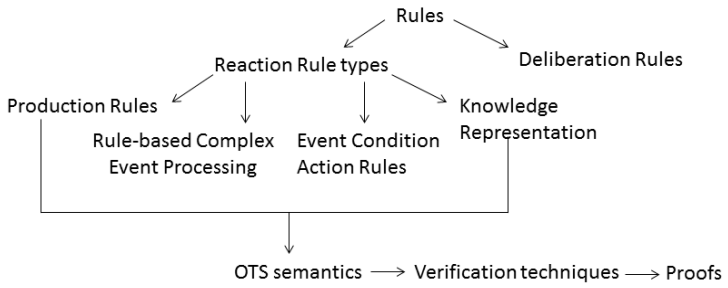


Figure 2: Reaction rules families and OTS

for all visible sort constants $d_{k_i} \in D_{k_i}$ such that there $\exists\, u \in Y$ for which c-$\tau_i(u, d_{1_i}, ..., d_{n_i}) = true$ we define:

- a hidden sub-sort, $h_{id_{1_i} \ldots d_{n_i}} \leq H$. Such that $\forall u \in h_{id_{1_i} \ldots d_{n_i}}$ we have that c-$\tau_i(u, d_{1_i}, \ldots, d_{n_i}) = true$. This is the hidden sub-sort that satisfies the effective condition of the transition rule τ_i for visible sort arguments $d_{1_i}, \ldots, d_{n_i}$.
- also for each transition rule $\tau_i : HD_{1_i} \ldots D_{n_i} \rightarrow H$ we define a new transition $\tau'_{id_{1_i} \ldots d_{n_i}} : h_{id_{1_i} \cdots n_i} \rightarrow H$, such that $\tau'_i(u) = \tau_i(u, d_{1_i}, \ldots, d_{n_i})$ $\forall u \in h_{id_{1_i} \ldots d_{n_i}}$.

We will call the OTS $S' = \langle O', I', T' \rangle$ defined from $S = \langle O, I, T \rangle$ as

- $O' = O$
- $T' = \{\tau'_i | \tau_i \in T\}$
- $I' = I$

the production OTS of S .

For a given state of the system $u \in Y$ it is possible to decide if it belongs to a sort or not, by checking whether $\exists\, d_{i_k}$ such that c-$\tau_i(u, d_{i_1}, \ldots, d_{i_n}) =$ true. If the previous expression holds then $u \in h_{id_{i_1} \ldots di_n}$. The next definition describes how with the above formalism we can semantically interpret a set of production rules as an OTS.

Definition 2. Assume now a set of production rules: *if* C_i *do* A_i, $i \in \{1, \ldots, N\}$. In such a set of rules C_i defines a set of constraints that when they hold the system should automatically apply the action(s) A_i. We define an OTS $S = \langle O, I, T \rangle$ such that the actions A_i's of the above rules are mapped to some transition rules τ_i's and the conditions C_i's to effective conditions c-τ_i's. The production OTS $S' = \langle O', I', T' \rangle$, created from S using definition 1 is the semantic interpretation of the production rules in the OTS framework.

Example 1. As an example of the previous definition consider the following production rule for an online store: *"If the status of a client is premium and the type of product is regular then assert discount of 25 percent for the customer"* [30]. It possible to fully characterize the above system with a set of queries (observers) as those in the table above, where *Client*, *Status*, *Product*, *Percent* and *Type* are appropriate predefined visible sorts. For this reason, we map the *assert* action to a transition rule, $assert : H\, Client\, Product \rightarrow H$. Given a client C and a product P, the effective condition for this transition rule is defined by the signature $c\text{-}assert$: $H\ Client\, Product \rightarrow Bool$ and the equation:

$$c - assert(H, C, P) = (client - status(H, C) = premium)\& \\ (product - type((H, P) = regular)$$

Now for each pair of constants c_i and p_i such that there exists a system state u that makes $c\text{-}assert(u, c_i, p_i) = true$, we define a new sub sort $h_{c_i p_i} \leq H$. Finally for each such sub sort we define a transition rule, $assert_i : h_{c_i p_i} \rightarrow H$, such that $\forall u \in h_{c_i p_i}$ we have; $assert_i(u) = assert(u, c_i, p_i)$. These transition rules can only be applied in states $u \in h_{c_i p_i}$, and in those states they are the only applicable transitions. So basically in an OTS containing only these transition rules we are enforcing the application of the desired transitions.

Following definition 2 we define the OTS $S' = \langle O', I', T' \rangle$ that corresponds to the production rule as follows:

- $O' = \{$*client-status*, *product-type*, *discount*$\}$

- I' the set of initial states

- $T' = \{assert_i \mid$ such that $h_{c_i p_i} \leq H\}$

In languages that implement OTS specifications however, definition 2 cannot be easily applied (possible infinity of sub-sorts). For this reason we present a 'specificational' approach to the semantics of production rules and prove that the two OTSs, S and S' defining the state spaces Y and Y' respectively, are behaviorally equivalent[1] [8].

In an OTS, a state is a kind of a black box. This means that each state is characterized only by the observable values returned by the observers $o \in O$. As a result, the effect of a transition rule (or a change of the state in general) can be characterized by the values returned by these observers. Now assuming a set of production rules of the form $if C_i$ *do* A_i,

[1] We will show that if S and S' have the same set of initial states and the same set of observers O then for $u \in Y$ and $u' \in Y'$ such that $\forall o \in O$, $o(u, y_1, \ldots y_n) = o(u', y_1, \ldots, y_n)$ applying a transition rule τ of S and τ' of S' will result in states $\tau(u), \tau'(u')$ such that $o(\tau(u), y_1, \ldots, y_n) = o(\tau'(u'), y_1, \ldots, y_n) \, \forall o \in O$.

Table 1: Observers of an OTS specifying a client discount system

Observer	Signature	Informal Definition
client-status	$H\,Client \rightarrow Status$	Returns the status of the client.
product-type	$H\,Product \rightarrow Type$	Returns the type of the product.
discount	$H\,Client\,Product \rightarrow Percent$	Returns the discount the client has on a product.

with $i \in \{1, \ldots, N\}$, we would ideally wish to correspond action A_i to a transition rule τ_i with an effective condition C_i. As we argued before it is not possible to enforce the application of a transition rule in an OTS. On the other hand it is possible to have conditional observations. So for an arbitrary state $u \in Y$ we have that after the application of the transition rule τ_i, $o_1(\tau_i(u, d_1, ..., d_n), v_1, ..., v_k) = v_1$ if $C_i(u, d_1, ..., d_n)$. These observations, led us to the following definition.

Definition 3. We define an OTS $S = \langle O, I, T \rangle$ from a set of production rules if C_i do A_i, with $i \in \{1, \ldots, N\}$ as:

- T = $\{read\}$, i.e. the only transition rule is $read : H \to H$, a single transition with no input.
- I is a set of initial states, such that $I \subseteq Y$.
- $O = \{o_1, \ldots, o_k\}$ is a finite set of observers. Each observer $o_i : H\, V_{i_1} \ldots V_{i_n}\ D_{i_1} \ldots D_{i_k} \to H$ satisfies the following equations for an arbitrary system state u:

$$o_i(read(u), v_{i_i}, ..., v_{i_n}, d_{i_1}, ..., d_{i_k}) = v_1 \ \textit{if}\ C_1(u, d_{i_1}, ..., d_{i_k})$$
$$o_i(read(u), v_{i_i}, ..., v_{i_n}, d_{i_1}, ..., d_{i_k}) = v_2 \ \textit{if}\ C_2(u, d_{i_1}, ..., d_{i_k})$$
$$\ldots$$
$$o_i(read(u), v_{i_i}, ..., v_{i_n}, d_{i_1}, ..., d_{i_k}) = v_N \ \textit{if}\ C_N(u, d_{i_1}, ..., d_{i_k})$$

Also $\forall d_{i_1}, \ldots, d_{i_k}, d'_{i_1}, \ldots, d'_{i_k}$ we have that $\forall o_i \in O, u \in I$:

$$o_i(u, v_{i_i}, \ldots, v_{i_n},\ d_{i_1}, \ldots, d_{i_k}) = o_i(u, v_{i_i}, \ldots, v_{i_n}, d'_{i_1}, \ldots, d'_{i_k})$$

This means that initially the values returned by the observers are only depended on the visible sorts $v_{i_i}, \ldots, v_{i_n}$ and not on the extra arguments $d_{i_1}, \ldots, d_{i_k}$ that are added to allow us to reason about the effective conditions of the transitions. Finally:

$$o_i(read(u), v_{i_i}, \ldots, v_{i_n}, d_{i_1}, \ldots, d_{i_k}) = o_i(u, v_{i_i}, \ldots, v_{i_n}, d_{i_1}, \ldots, d_{i_k})$$
$$if \neg(C_1(u, d_{i_1}, \ldots, d_{i_k}) \vee \cdots \vee C_N(u, d_{i_1}, \ldots, d_{i_k})$$

The intuition behind the previous definition is that we have a system that has one transition only. This transition basically checks to see if any of the conditions defined by the production rules are met and if so it automatically changes the values returned by the observers to those we would

expect if the corresponding to the condition action was applied.

Proposition 1. Two OTSs defined from a set of production rules R, using definitions 2 and 3 that have the same set of initial states are *behaviorally equivalent*, according to observational equivalence.

Proof. Since the two OTSs are equivalent for the initial states it remains to show the following:

($\rightarrow$) If we apply an arbitrary transition rule $\tau \in T_1$ to an arbitrary state $u \in Y_1$ such that u is behaviorally equivalent to a state of Y_2 then the only applicable transition from T_2 will lead us to a state that is behaviorally equivalent to $\tau(u)$. So assuming $u \in Y_1$ then we can decide if $\exists\, i, d_{i_1}, \ldots, d_{i_n}$ such as $u \in h_{id_{i_1},\ldots,d_{i_n}}$. We discriminate two cases.

1. If there are not such $i, d_{i_1}, \ldots, d_{i_n}$ then this means that there are no applicable transition rules in S_1 and that $\forall i, d_{i_1}, \ldots, d_{i_n}$ we have that $C_i(u, d_{i_1}, \ldots, d_{i_n}) = false$. But then by applying the only transition rule of S_2 to $u' \in Y_2$ (the behaviorally equivalent state of u) we will have that $\forall o_{i2} \in O_2, \forall d_{i_1}, \ldots, d_{i_k}$; $o_{i2}(read(u'), v_{i_1}, \ldots, v_{i_n}, d_{i_1}, \ldots, d_{i_k})$ $= o_{i2}(u', v_{i_1}, \ldots, v_{i_n}, d_{i_1}, \ldots, d_{i_k})$. So u' and read(u') are behaviorally equivalent, but then so is read(u') with the state of S_1 u that remains unchanged because there is no applicable transition in S_1 for u.

2. If there exists $i, d_{i_1}, \ldots, d_{i_n}$ such that $u \in h_{id_{i_1},\ldots,d_{i_n}}$, this means that the only applicable transition for S_1 is $\tau_{id_{i_1},\ldots,d_in}$. This implies that $C_i(u, d_{i_1}, \ldots, d_{i_n}) = true$, and also $\forall o_{j1} \in O_1$ we have that $o_{j1}(\tau_{id_{i_1},\ldots,d_{i_n}}$ $(u), v_{j_1}, \ldots, v_{j_n}) = v_{j_i} = o_{j1}(\tau_i(u, d_{i_1}, \ldots, d_{i_n}), v_{j_1}, \ldots, v_{j_n})$. But now since $C_i(u, d_{i_1}, \ldots, d_{i_n}) = true$, in S_2 we have that $\forall o_{j2} \in O_2$, after the application of the only transition rule $read(u)$, $o_{j2}(read(u), v_{j_1}, \ldots, v_{j_n}$, $d_{i_1}, \ldots, d_{i_n}) = v_{j_i}$. So $read(u)$ and $\tau_{id_{i_1},\ldots,d_in}(u)$ are behaviorally equivalent since all the corresponding observers return the same values.

($\leftarrow$) It remains to show that by applying an arbitrary transition on S_2 we get a behaviorally equivalent state in S_1. The only transition we can apply in S_2 is $read()$. Then $\forall o_{j2} \in O_2$ if $o_{j2}(read(u), v_{j_1}, \ldots, v_{j_n}, d_{i_1}, \ldots, d_{i_n}) = v_{j_i}$ then $C_j(u, d_{i_1}, \ldots, d_{i_n}) = true$. But then in S_1 we will have that $u \in h_{id_{i_1},\ldots,d_in}$. But from definition 1 the only transition allowed to u is $\tau_{id_{i_1},\ldots,d_{i_n}}$. Then $o_{j1}(\tau_{id_{i_1},\ldots,d_{i_n}}, v_{j_1}, \ldots, v_{j_n}) = o_{j1}(\tau_i(u, d_{i_1}, \ldots, d_in), v_{j_1}, \ldots, v_{j_n}) = v_{j_i}$. Meaning that the states are behaviorally equivalent.

3.1.2 Event Condition Action (ECA) Rules and OTS semantics

Event Condition Action rules (ECA) are one of the most commonly used categories of reactive rules. Their syntax is: *on Event if Condition do Action*, where event denotes an explicit action that changes the state of the system, and action denotes a change in the state of the system that is caused as a reaction to the event, if the condition part of the rule holds.

Definition 3. The concept of an ECA rule, $r = On\ E_i\ if\ C_i\ do\ A_i$, can be transferred naturally in OTS notation. Suppose transition rules $\tau : S\ D_1 \dots D_n \to S$ and $\tau' : S\ D'_1 \dots D'_k \to S$, that are under the effective conditions $c\text{-}\tau$ and $c\text{-}\tau'$ respectively. We assume that the first rule specifies the state change due to E_i and the second rule specifies the state change due to A_i. We also assume that $c\text{-}\tau'$ corresponds to C_i. We map r to a new transition rule in the OTS $r : SD_1 \dots D_n D'_1 \dots D'_k \to S$. Such that for an arbitrary state u and for all observers o that change their returned value when τ' is applied we have: if $o(\tau'(u, d'_1, \dots, d'_k), v_1, \dots, v_m) = v_o$ when $c\text{-}\tau'(u, d'_1, \dots, d'_k)$, then $o(r(u, d_1, \dots, d_n, d'_1, \dots, d'_k)\ v_1, \dots, v_m) = v_o$ when $c\text{-}\tau'(\tau(u, d_1, \dots, d_n), d'_1, \dots, d'_k)$ and $c\text{-}\tau(u\ , d_1\ , , d_n)$. For all observers o' whose observations remain unaffected by the transition rule τ' we define; if $o'(\tau(u, d_1, , d_n), v'_1, , v'_q) = v'_o$ then:

$$o'(r(u, d_1, \dots, d_n, d'_1, \dots, d'_k)v_1, \dots, v_m) = v'_o$$
$$if\ c - \tau'(\tau(u, d_1, \dots, d_n), d'_1, \dots, d'_k)\ \&\ c - \tau(u, d_1, \dots, d_n)$$

This transition rule r, basically defines the sequential application of transition rules τ and τ'.

Example 2. As an example assume the following ECA rule: *"On receiving premium notification from marketing and if regular derivable do send discount to customer"* [30]. We identify the following components in that rule: The event *receiving premium notification from marketing*, the condition *regular derivable* and the action *send discount to customer*. By mapping the event and action to transitions and using the appropriate observers (tables 2 and 3) we can define an OTS S corresponding to this ECA rule.

According to definition 3, we can use the previous OTS to define an OTS $S' = \langle O', T', I' \rangle$ that models the ECA rule as follows:

- $O' = O$, where $O = \{customer\text{-}type, discount, product\text{-}type\}$
- $I' = I$, where I is the set of initial states of S
- $T' = \{t\}$, where $t : S\ SenderId\ ClientId\ ProdId \to S$, represents the sequential application of the transitions $receive\text{-}notification$ and $send\text{-}discount$.

The result of applying t to an arbitrary system state S is defined by the following equations:

discount$(t(S, I1, C1, P1)), P2, C2)$ = *true if c-receive-notification*$(S, I1, C1)$ $\wedge$ *product-type(receive-notification*$(S, I1, C1), P1))$ = *regular* $\wedge$ $(C1 = C2)$ $\wedge$ $(P1 = P2)$

customer-type$(t(S, I1, C1, P1), C2)$ = *customer-type(receive-notification* $(S, I1, C1), C2)$

product-type$(t(S, I1, C1, P1), P2)$ = *product-type*$(S, P2)$

In the equations above $I1$ is a variable denoting an arbitrary sender, $C1$, $C2$ are variables denoting arbitrary clients and finally $P1$, $P2$ are variables denoting arbitrary products.

In definition 3 however, we define a transition rule that contains, on the effective condition, a reference to a successor state of the arbitrary system state u, namely $\tau(u, d_1, \ldots, d_n)$. This approach while providing clear and intuitively straight semantics for ECA rules cannot be applied to the algebraic specification languages that implement OTSs. The reason for this is that it is not permitted to have a transition rule on the right hand side of an equation defining another transition rule. This guarantees the termination of the rewriting procedure that is used to create proofs with such languages. So once again we will define another (semantically equivalent) model for the ECA rules that is more specification orientated than intuitively straight forward. We wish to have transitions (the events) that occur from an outside source to the system like the typical transitions of an OTS, but at the same time we require for our system to react to these events if some conditions hold. We try to achieve this double goal by allowing the systems refresh transition (read) that we defined for the production rules to be parameterized. Also we modify the OTS, with a memory observer, so that it remembers if in the previous state an event occurred or not. The

Table 2: Transitions of an OTS specifying a notification client discount system

Transitions	Signature	Informal Definition
receive-notification	$S\, SenderId\, ClientId \rightarrow S$	Models the fact that a premium notification for a client has been sent by a sender
send-discount	$S\, ProdId\, ClientId \rightarrow S$	Models the fact that a discount for a client on a product is granted

Table 3: Observers of an OTS specifying a notification client discount system

Observers	Signature	Informal Definition
customer-type	$S\,ClientId \rightarrow Status$	Returns the status of the given client
discount	$S\,ProdId\,ClientId \rightarrow Bool$	Returns true if the customer has discount on a product
product-type	$S\,ProdId \rightarrow Type$	Returns the type of a product (regular or not)

parameterization allows us to simulate the execution of events, while the memory allows us to decide if the OTS must react to this refresh or treat it as an incoming event.

Assume a finite set of ECA rules $\{r_i = On\ E_i\ if\ C_i\ do\ A_i \mid i \in \{1, \ldots, n \in \mathbb{N}\}\}$ and without harm of generality assume that for $i \neq j; E_i, A_i \neq E_j, A_j$ respectively. Also assume that for these events and actions there exist predefined transition rules in an OTS say S and that the visible sorts $D_1, \ldots, D_l$ were required for their definition.

Definition 4. We define a new OTS $S' = \langle O', I', T' \rangle$ modeling these rules, where:

- $O' = O \cup \{memory\}$. Memory is a special observer that remembers if an event has occurred in a state and what that event was. Since the set of rules is finite we have a finite set of events. We can now define the observer $memory : S\,D_1 \ldots D_l \rightarrow \{1, \ldots, n \in \mathbb{N}\}$.

- $I' = I$.

- $T' = \{read\}$. Where $read : S\,\{1, \ldots, n \in \mathbb{N}\} \rightarrow S$, a single parameterized transition function. This according to the value of the index $n \in \mathbb{N}$ models the transition that corresponds to event E_n. For an arbitrary system state u and $i \in \{1, \ldots, n \in \mathbb{N}\}$ we define that $\forall o \in O$:

$$o(read(u,i), d_1, \ldots, d_l, v_1, \ldots, v_q) = v_{E_i}\ if\ c - \tau_i(u, d_1, \ldots, d_l) \\ \&\ (memory(u, d_1, \ldots, d_l) = null)$$

  This equation states that o will return the same value as it would in S, when the transition (event) E_i had successfully occurred, if the memory is empty. The memory is empty at the initial states and after the occurrence of an action. In the case where the memory is not empty we specify that this triggers a reaction from the OTS with the following equation:

$$o(read(u,i),d_1,\ldots,d_l,v_1,\ldots,v_q) = v_{A_i} if c - \tau_{A_i}(u,d_1,\ldots,d_l) \\ \& (memory(u,d_1,\ldots,d_l) = i)$$

This equation states that o will return the same value as it would in S, when the transition (action) A_i had occurred successfully, if the memory contains the index i (i.e. in the previous state of S we had an occurrence of the event i).

Revisiting example 2, we can define the OTS $S' = \langle O', I', T' \rangle$ according to definition 4, as follows:

- $O' = \{customer\text{-}type, discount, product\text{-}type, memory\}$
- $T' = \{read\}$
- $I' = I$

Since in this example we have one event only, we map it to index 1. So for an arbitrary system state S the effect of $read(S,1)$ on S is defined by the values returned by the observers given in the following equations:

customer-type(read(S, 1), C1) = premium if c-receive-notification(S) & memory(S) = null.

memory(read(S, 1)) = 1 if memory(S) = null.

discount(read(S, 1), P1, C1) = true if product-type(S, C1) = regular & memory(S) = 1.

memory(read(S, 1)) = null if memory(S) ≠ null.

It is important to mention that in the case of multiple ECA rules, the semantics of events is non-deterministic, i.e. in an arbitrary state arbitrary events can be applied. On the other hand, actions are translated to deterministic behavior.

3.1.3 Complex Events and OTS semantics

We define the semantics of a complex event algebra as an algebra for a (timed) OTS. We chose the event algebra of [31] for reference. We will define the semantics for some standard event algebra operators. In order to do this however we must first introduce the notion of an observer group in a (timed) OTS.

Definition 5. Assuming a (timed) OTS S, we define that the transition $\tau \in T$ belongs to the Observer Group $og = \{o_1, , o_n\} \subseteq O$ iff $\forall o \in O \setminus og$, $o(\tau(u, d_1, , d_n), v_1, \ldots, v_n) = o(u, v_1, \ldots, v_n)$.

Now assume that in our OTS the semantics of primitive events are those of transitions, meaning that each primitive event A, is mapped to a transition rule in the OTS. The proposed event algebra of [31] consists of the following complex event operators; *disjunction*, *conjunction*, *negation*, *sequence* and *temporal restriction*. In the following definition we specify the semantics of these operators in the OTS framework inductively.

Definition 6. Each transition rule that denotes a primitive event is a complex event. Assuming complex events A and B with effective conditions c_A and c_B respectively we define the following transition rules:

The *disjunction* transition rule $A \vee B : S\, D_{A_1} \ldots D_{A_n} D_{B_1} \ldots D_{B_m} \rightarrow S$, with effective condition $c_A \vee c_B$. Where $\forall o \in O$:

$o(A \vee B(u, d_{A_1}, \ldots, d_{A_n}\, d_{B_1}, \ldots, d_{B_m}), v_{o_1}, \ldots, v_{o_k}) = o(A(u, d_{A_1} \ldots d_{A_n}), v_{o_1}, \ldots, v_{o_k})$ if c_A *or*

$o(A \vee B(u, d_{A_1}, \ldots, d_{A_n}\, d_{B_1}, \ldots, d_{B_m}), v_{o_1}, \ldots, v_{o_k}) = o(B(u, d_{B_1} \ldots d_{B_n}), v_{o_1}, \ldots, v_{o_k})$ if c_B.

Meaning that either event A happens or event B, but not both.

The *sequence* transition $A; B : S\, D_{A_1} \ldots D_{A_n}\, D_{B_1} \ldots D_{B_m} \rightarrow S$ as the composition of transitions A and B. Such that $\forall o \in O$:

$o(A; B(u, d_{A_1} \ldots d_{A_n}\ d_{B_1} \ldots d_{B_m}),\ v_{o_1}, \ldots, v_{o_k}) = o(A(B(u, d_{A_1} \ldots d_{A_n}) , d_{B_1}) \ldots d_{B_m}) , v_{o_1}, \ldots, v_{o_k})$.

With effective condition $c_{A;B}(S) = c_B(S) \wedge c_A(S')$, where S' is the successor state of S when transition B is applied to it.

The *conjunction* transition rule $A + B$, denoting that both events occur. If the events occur simultaneously then for the system to be able to observe them they have to belong to different observer groups. So we define $A + B : S\, D_{A_1} \ldots D_{A_n} D_{B_1} \ldots D_{B_m} \rightarrow S$, such that $\forall o \in og_A, o \in og_B$:

$o(A + B(u, d_{A_1} \ldots d_{A_n}\ d_{B_1} \ldots d_{B_m}), v_{o_1}, \ldots, v_{o_k}) = o(A(u, d_{A_1} \ldots d_{A_n}), v_{o_1}, \ldots, v_{o_k})$ *and*

$o(A + B(u, d_{A_1} \ldots d_{A_n}\ d_{B_1} \ldots\ d_{B_m}), v_{o_1}, \ldots, v_{o_k}) = o(B(u, d_{B_1}\ \ldots d_{B_n}),$

$v_{o_1}, \ldots, v_{o_k}$).

Where og_A and og_B are the observer groups of transitions A and B respectively. Note that $og_A \cap og_B = \emptyset$. If the events do not occur simultaneously then $A + B$ and $B + A$ are equivalent to the sequential complex events $A; B$ and $B; A$, respectively.

The *negation* transition A-B that denotes the state where there is an occurrence of event A while event B does not occur. Occurrence of an event A denotes that the observers effected by transition A have the same values as when A is applied to an arbitrary state. In this case we wish to define a new transition rule stating that while all the observers in the observer group of A return the same values as those returned by applying A to an arbitrary state it is not possible for event B to occur. Since we are in a Timed OTS (TOTS) we have two associated observers, l_τ and u_τ for each transition τ denoting the lower and upper time bound of the transition rule respectively. So it suffices to define $A\text{-}B : S\, D_{A_1} \ldots D_{A_n} \to S$ such that $\forall o \in O$:

$o(A\text{-}B(u, d_{A_1} \ldots d_{A_n}), v_{o_1}, \ldots, v_{o_k}) = o(A(u, d_{A_1} \ldots d_{A_n}), v_{o_1}, \ldots, v_{o_k})$ & $l_B(A\text{-}B(u, d_{A_1} \ldots d_{A_n})) = \infty$.

Finally, we define *temporal restrictions*, i.e. an occurrence of an event A shorter than $\tau - time$ units. In the TOTS framework the effective condition of transitions *tick* basically forces the time to stop advancing if it will surpass the upper bound of any transition rule. So we define the complex event $A\text{-}time : SD_{A_1} \ldots D_{A_n}\mathbb{R}^+ \to S$ such as $\forall o \in og_A$:

$$o(A\text{-}time(u, d_{A_1} \ldots d_{A_n}, \tau), v_{o_1}, \ldots, v_{o_k}) = o(A(u, d_{A_1} \ldots d_{A_n}), v_{o_1}, \ldots, v_{o_k}),$$

under the same effective condition as transition A.

Also for all other primitive or complex events τ that belong to the same observer group as A, we define that:

$$l_\tau(A - time(u, d_{A_1} d_{A_n}, \tau)) = now(A - time(u, d_{A_1} \ldots d_{A_n}, \tau)) \; and$$

$$u_\tau \; (A - time(u, d_{A_1} \ldots d_{A_n}, \tau)) = now(A - time(u, d_{A_1} \ldots d_{A_n}, \tau)) + \tau.$$

The above equations ensure that A will no longer occur after τ-time units, since the clock will be stopped until a transition of the same observer group as A is applied successfully.

3.1.4 Knowledge Representation Rules and OTS semantics

Knowledge representation (KR) focuses on the inferences that can be made from the fact that certain events are known to have occurred or are planned to happen in future. Among the KR formalisms, are the Event Calculus (EC) [32], the Situation Calculus (SC) [33], various action languages and event logics. Here we will focus on the first two approaches.

In *SC approach* a set of properties of interest for the system, say $\{P_1, \ldots P_n\}$ is assumed. For an arbitrary system state S and an action of the system A, it is defined that if action A occurs in situation S a new situation results ($result(A, S)$). In ($result(A, S)$) property $P \in \{P_1, \ldots P_n\}$ will be *true* (*false*) if action A in state S *initiates* (*terminates*) P. Let us also mention here that with SC we mean the original version of McCarthy [33] and not of R. Reiter, i.e. a situation is a state or a snapshot rather than a sequence of actions.

Definition 7. The formalization of such a system in the OTS approach can be done using the OTS $S = \langle O, I, T \rangle$ where:

- $T = \cup\{A_i\}$ a finite set of transitions that correspond to the finite set of actions defined by the rules. Each such transition is defined as $A_i : Sys\, D_1 \ldots D_{k_i} \rightarrow Sys$.
- I a set of initial states.
- O is the set of observers $\{initiated, terminated\} \cup \{P_i\}$.

Where $initiated : Sys\, Label1\, Label2 \rightarrow Bool$, $terminated : Sys\, Label1\, Label2 \rightarrow Bool$ and $P_i \in \{P_1, \ldots P_n\}$ with $P_i : Sys\, D_{1_i} \ldots D_{n_i} \rightarrow Bool$. Also $Label1$ and $Label2$ are predefined visible sorts that denote the actions of the system and the properties of interest, respectively. The first observer returns true when action A initiates property P and the second observer returns true when action A terminates property P. This is formalized by the following equations;

$$P_j(A_k(S, v_{1_i}, \ldots, v_{n_i})) = true \text{ if } c\text{-}A_k(S) \wedge initiated(S, p_j, a) \;\&$$

$$P_j(A_k(S, v_{1_i}, \ldots, v_{n_i})) = false \text{ if } c\text{-}A_k(S) \wedge terminated(S, p_j, a).$$

Where, the constants a, p_j denote an arbitrary action and a property j, respectively, while $v_{1_i} \ldots v_{n_i}$ denote arbitrary visible sorts values needed for the definition of the transitions.

In *EC approach* a model of change is defined in which events happen at time points and initiate and/or terminate time intervals over which certain properties of the world hold. The basic idea is to state that properties are true at particular time points if they have been initiated by an event at some

earlier time point and not terminated by another in the meantime.

Definition 8. This notion can be formalized in the OTS approach as well using a TOTS $S = \langle O, I, T \cup \{tick_r\}\rangle$ defined as follows.

- T is the set of transitions such that $T = \cup\{A_i\}$. Where $A_i : Sys \rightarrow Sys$ are a finite set of transitions that correspond to the finite set of actions defined by the rules and $\{tick_r\}$ is the usual advancing time transition defined in the generic TOTS.
- I is a set of initial states.
- O the set of observers, where $O = \{initiated, terminated, now\} \cup \{P_i\}$, where *initiated*, *terminated* and P_i are defined as in the previous definition. Also *now* is a special observer whose signature is $now : S \rightarrow \mathbb{R}^+$ and denotes the system's master clock. Finally, the equations defining the observers in an arbitrary state (adopting the same notation as definition 7) are the following:

$P_j(A_k(S, v_{1_i}, \ldots, v_{n_i})) = true$ if $\exists\, S', A_m$ such that $initiated(S', p_j, a_m) = true$ & $now(S') \leq now(S)$ *or*

$P_j(A_k(S, v_{1_i}, \ldots, v_{n_i})) = true$ if $\nexists\, S', A_m$ such that $terminated(S', p_j, a_m) = true \wedge now(S') \leq now(S)$.

Also:

$P_j(A_k(S)) = false$ if $\nexists\, S'', A_m$ such that $(initiated(S'', p_j, a_m) = true$ & $now(S'') \leq now(S)$ *or*

$P_j(A_k(S)) = false$ if $\exists\, S'', A_m$ such that $(terminated(S'', p_j, a_m) = true$ & $now(S'') \leq now(S)$.

In languages implementing OTSs quantifiers need to be treated carefully. This is due to the fact that these languages usually relay on equational logic. There, the $\forall$ quantifier can be handled by free variables, i.e. each equation $E(x)$ containing an unbound variable x is semantically equivalent to $\forall x E(x)$. On the other hand the $\exists$ quantifier is not straightforwardly supported. However, each equation containing an $\exists$ quantifier can be transformed into its equivalent Skolem normal form without such quantifiers.

Definition 9. The equations defining the observers in the above definition, can be replaced with the following for a language that implements an OTS.

$P_j(A_k(S, v_{1_i}, \ldots, v_{n_i})) = true$ if $initiated(f_{S'}(S), p_j, f_{a_m}(S)) =$ true $\wedge$ $now\ (f_{S'}(S)) \leq now(S))$ *or*

$P_j(A_k(S, v_{1_i}, \ldots, v_{n_i})) = true$ if $terminated(f_{S'}(S), p_j, f_{a_m}(S)) = true$ $\wedge\ now(f_{S'}(S)) \leq now(S))$.

Also:

$P_j(A_k(S)) = false$ if $(initiated(f_{S'}(S), p_j, f_{a_m}(S)) = true \wedge now(f_{S'}(S))$ $\leq now(S)\ )$ *or*

$P_j(A_k(S)) = false$ if $(terminated(f_{S'}(S), p_j, f_{a_m}(S)) = true \wedge now(f_{S'}(S))$ $\leq now(S))$.

Where $f_{S'}$ and f_{a_m} are the Skolemization functions that map each hidden sort to a hidden sort and each hidden sort to a label, respectively. The formalization for EC presented here corresponds to Simplified Event Calculus (SEC), i.e. we employ time points instead of time periods. A similar approach however could be adopted for the original EC as well.

3.2 An Algebraic Framework for Modeling of Reactive Rule-based Intelligent Agents

In this section, we address the problem of formally analyzing reactive rules, by presenting a methodology based on the OTS/CafeOBJ method that defines their behavior in terms of equational transition rules. The proposed framework offers in this way the ability to formally specify an intelligent agent whose behavior is expressed in terms of reactive rules, to verify its behavior and thus ensure its correctness. This is in continuation of the previous section and of [35], where Observational Transition System (OTS) semantics were provided for reactive rules. However here we adapt and improve the proposing approach so that it can be easily used in the context of the algebraic specification language CafeOBJ. In order to demonstrate its effectiveness, we apply the framework to a supply chain management system and prove security properties about the system.

Intelligent agents are a new paradigm for developing software applications. An intelligent agent is defined either as anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors [36], or as a software that carries out some set of operations and acts on behalf of a user [37], or finally as a computational process that implements the autonomous functionality of an application [38]. Agent-based systems usually consist of many agents that communicate with each other and are known as multi-agent systems.

The use of rule-based systems as the main reasoning model of agents that are part of a multi-agent system has been proposed in early attempts. In this approach each agent includes a rule engine and is able to perform rule-based inference [39]. Thus, an intelligent agent is called rule-based, if its behavior and its knowledge are expressed by means of rules.

The task of verifying the behavior of rule-based agents is difficult because rules can interact during execution and this interaction can cause undesirable results [40]. For example, one rule may trigger another rule and cause a chain of rule triggering. Also, changes to the rule base (add, remove, change rules) can introduce errors in the behavior of the system if the effects of the changes are not examined beforehand. Thus, using rules in critical systems implies that the system's behavior must be extensively analyzed.

The proposed framework supports Event Condition Action and Production rules. This allows proving desired safety properties about intelligent systems whose behavior is expressed in terms of such rules. Because we are interested in proving application specific properties, additional characteristics (observers and/or transitions) about the specific system will be required in order to specify its behavior. These characteristics will differ from application to application and thus the specification cannot become fully automated. This framework however will serve as the basis for specifying and verifying reactive rule based systems and most importantly for

capturing the semantics of their rules.

3.2.1 Production Rules in CafeOBJ

A Production rule is a statement of rule programming logic, which specifies the execution of an action in case its conditions are satisfied, i.e. production rules react to states changes. Recall that their essential syntax is *if Condition do Action*. Some usual predefined actions supported by Rule Markup languages are: add, retract, update knowledge and generic actions with external effects [41].

A Production rule can be naturally expressed in our framework if we map the action of the rule to a transition which has as effective condition the condition of the rule. Also, since most of the actions correspond to changes of the knowledge base in order to describe their effects we need an observer that will observe the knowledge base (KB) at any given time. Thus, the observer $knowledge : Y \rightarrow SetofBool$ which returns the set of boolean elements that belong to the knowledge base is needed. For expressing the functionalities of the KB, the following operators are required; $/in$ which returns true if an element belongs to the knowledge base, | which denotes that an element is added to the KB and / which denotes that an element is removed from the KB. Formally, the definition of a set of Production rules as an OTS is presented below.

Definition 10. Assume the universal state space Y and the following set of Production rules; {if C_i do A_i, $i = 1, \ldots, n \in N$}, where without harm of generality we also assume that the conditions of the rules are disjoint. We define an OTS $S = \langle O, I, T \rangle$ from this set of rules as follows:

- $O = \{O' \cup knowledge\}$
- T = $\{A_i\}$
- I = the set of initial states, such that $I \subseteq Y$

In the above definition, O' denotes the rest of the system's observers. Transitions are the actions of the rules, $A_i : YD_1 \ldots D_l \rightarrow Y$. They can be generic actions (with external changes) or the usual predefined actions $assert : YBool \rightarrow Y$ (add a fact to KB), $retract : YBool \rightarrow Y$ (remove a fact from KB), $update : YBool\,Bool \rightarrow Y$ (remove/add a fact) [42]. Facts are denoted by boolean-sorted CafeOBJ terms. Formally, the actions of Production rules are defined as transitions (in CafeOBJ terms) through the following steps;

1. The effective condition of an action A_i is defined in CafeOBJ terms as;

```
c-Ai(u,d1,...,dn) = Ci(d1,...,dn) /in knowledge(u).
```

2. If A_i is an assert action its effect on the knowledge observer is defined as;

```
knowledge(assert(u,ki(d1,...,dn))) = ki(d1,...,dn)
|knowledge(u) if c-assert(u,ki(d1,...,dn)).
```

3. If A_i is a retract action its effect on the knowledge observer is defined as;

```
knowledge(retract(u,ki(d1,..,dn))) = knowledge(u)
/ki(d1,...,dn) if c-retract(u,ki(d1,..,dn)).
```

4. If A_i is an update action its effect on the knowledge observer is defined as;

```
knowledge(update(ki(d1,..,dn),kj(d1,..,dn))) = (knowledge(u)
/ki(d1,..,dn))|kj(d1,..,dn) if c-update(ki(d1,..,dn),kj(d1,
...,dn)).
```

5. If A_i is a generic action, we define;

```
knowledge(ai(u,d1,...,dn)) = ai(d1,...,dn)|knowledge(u)
if c-ai(u,d1,...,dn).

oi(ai(u,d1,...,dn)) = vi if c-ai(u,d1,..,dn).
```

Step 1 declares that an action a_i can be successfully applied if the condition of the rule holds, i.e. belongs to the knowledge base[2]. Step 2 states that when a transition $assert(u, k_i(d_1, \ldots, d_n))$ is applied successfully in an arbitrary state u, k_i is added to the knowledge base. Where k_i is the fact being asserted. In step 3 it is stated that when the transition $retract(u, k_i(d_1, \ldots, d_n))$ is applied successfully in an arbitrary state u, k_i is removed from the knowledge base. When the transition $update(u, k_i(d_1, \ldots, d_n), k_j(d_1, \ldots, d_n))$ is applied successfully in an arbitrary state u, ki is removed and kj is added, as step 4 defines. Step 5 states that when we have the application of a generic action we add to our KB the information that this action occurred. But generic actions may have side effects and in order to describe them we may have to use additional observers $o_i \in O$ and define how their values change when the action is applied successfully. Finally, if the effective condition of a transition does not hold, the state of the system remains the same (this is the reason we do not need to use the refresh transition *read* we had introduced in [35])

[2]If we have negation-as-failure in the condition of the rule, i.e. if the condition cannot be proved, this is expressed in our framework as; if $c_i \notin knowledge(u)$, since this basically means that there is no information (in our knowledge base) about the condition.

3.2.2 Event Condition Action Rules in CafeOBJ

In contrast to Production rules, Event Condition Action (ECA) rules define an explicit event part which is separated from the conditions and actions of the rule. Recall that their essential syntax is; *on Event if Condition do Action*. The ECA paradigm states that a rule autonomously reacts to actively or passively detected simple or complex events by evaluating a condition or a set of conditions and by executing a reaction whenever the event happens and the condition(s) is true [43].

In order to express ECA rules in our framework we need an observer that will remember the occurred events. For this reason, in each event we assign a natural number and when an event is detected its number is stored in the observer $event\text{-}memory : Y \rightarrow Nat$. Using *event-memory* we can map events to transitions. The actions of ECA rules are assert, retract, update, or generic actions and are mapped to transitions, as before. However, their semantics differs as the actions of ECA rules can be applied only if their triggering event has been detected first. Formally, the definition of a set of ECA rules as an OTS is presented below.

Definition 11. Assume the universal state space Y and a finite set of ECA rules $\{\text{on } E_i \text{ if } C_i \text{ do } A_i, i = 1, \ldots, n \in N\}$, where without harm of generality we also assume that for $i \neq j$; $E_i, A_i, C_i \neq E_j, A_j, C_j$, respectively. The OTS $S = \langle O, I, T \rangle$ modeling these rules is defined as:

- $O = \{O' \cup knowledge, event - memory\}$
- $T = \{E_i, A_i\}$
- $I =$ the set of initial states such that $I \subseteq Y$

Here, O' is the same as in definition 10. Transitions are the events, $E_i : Y D_1 \ldots D_n \rightarrow Y$ and the actions, $A_i : Y D_1 \ldots D_n \rightarrow Y$. Formally, the rule on E_i if C_i do A_i is defined in CafeOBJ terms through the following steps:

1. The effective condition of an event E_i is denoted as $c\text{-}ei(u, d1, \ldots, dn)$ and states the conditions under which the system is able to detect the event.

2. The effects of the application of the event E_i in an arbitrary system state u are the following;

```
knowledge(Ei(u,d1,...,dn)) = ei(d1,...,dn)|knowledge(u) if
c-ei(u,d1,...,dn) and event-memory(u) = null .

event-memory(Ei(u,d1,...,dn)) = i if c-ei(u,d1,...,dn) and
event-memory(u) = null .
```

3. The effective condition of the action A_i, is defined as;

```
c-Ai(u,d1,...,dn) = Ci(d1,...,dn) /in knowledge(u) and
ei(d1,...,dn) /in knowledge(u).
```

4. The effects of the action A_i, if it is an assert action, are described through the following equations;

```
knowledge(assert(u,ki(d1,..,dn))) = ki(d1,..,dn)|knowledge
(u)/ei(d1,...,dn) if c-assert(u,ki(d1,...,dn)).

event-memory((u,ki(d1,...,dn))) = null if c-assert(u,ki(d1
,...,dn)).
```

The effects of the rest of the actions are defined in a similar way. Step 2 states that when the transition/event E_i is applied, the name of the occurred event (e_i) is added to the knowledge base as a fact if in the previous state the detection conditions of the event were true and event-memory was null (denoting that no events had occurred). Also, when the event is applied, event-memory stores the identification number of the event (here i). Step 3 declares that the action will be applied successfully, if the condition of the rule belongs to the KB and the triggering event of the action has been detected. In step 4 it is stated that when the action $assert(u, ki(d1, \ldots, dn))$ is applied, the fact k_i is added to the knowledge base and its triggering event is consumed, i.e. its name is removed from the knowledge observer. Also, *event-memory* becomes null.

We must mention here that as we will see in the following section, sometimes the names of the events are not removed from the observer event-memory if they are required for the detection of complex events. Also, if many rules (either Production or ECA) can be executed at the same time, a selection function is used from the inference engine of the system such as those presented in [44, 45]. It is quite straightforward to include this characteristic in our framework but is out of the scope of this chapter.

One of the challenges we met while expressing these rules into our framework was the difference between events and actions, i.e. while events *can* occur at anytime and can be straightforwardly mapped to transitions, actions *must* be executed after the detection of their triggering events. To capture this difference we used the observer event-memory. Initially it returns the value null (meaning that no events have been detected) denoting that any event can occur, but when an event is detected then the only applicable transition in the system is the action of the detected event.

3.2.3 Complex Events and CafeOBJ

Sometimes ECA rules react to the detection of complex events. Complex events are created by primitive event(s) and event operator(s). A typical

set of event operators for defining complex events include the following; xor (mutually exclusive), disjunction (or), conjunction (and), any, concurrent (parallel), sequence (ordered), aperiodic, periodic. In [44] definitions of such operators are presented in more details. In this section we will present how the basic event operators can be expressed in our framework.

Definition 12. Assume primitive events A_i and B_j defined as transitions with effective conditions c-A_i and c-B_j respectively. Complex event $xor(Ai, Bj)$ means that either event Ai happens or Bj, but not both. The application of the complex event/transition $e_k : xor(u, Ai, Bj)$ to an arbitrary system state is defined as:

```
knowledge(xor(u,Ai,Bj)) = xor(Ai,Bj)|knowledge(u) if Ai /in
knowledge(u) xor Bj /in knowledge(u) .

event-memory(xor(u,Ai,Bj)) = k if Ai /in knowledge(u) xor Bj
/in knowledge(u) .
```

The above equations state that the complex event is detected (its occurrence is added to the KB) if its detection conditions are fulfilled, i.e. if we have detected either the primitive event Ai or event Bj. Also, the observer event-memory stores the id number k of the event (where xor is a built-in operator) if the same conditions hold.

$Disjunction(Ai, Bj)$ means that either event Ai happens or Bj (or both). In a similar way, the application of the event $disjunction(u, Ai, Bj)$ is defined as;

```
knowledge(disjunction(u,Ai,Bj)) = disjunction(Ai,Bj)|
knowledge(u) if Ai /in knowledge(u) or Bj /in knowledge(u).
```

$Conjunction(Ai, Bj)$ means that both events Ai and Bj occur in any order. The application of the event $conjunction(u, Ai, Bj)$ is defined as;

```
knowledge(conjuction(u,Ai,Bj)) = conjuction(Ai,Bj)|knowledge(u)
if Ai /in knowledge(u) and Bj /in knowledge(u).
```

$Sequence(Ai, Bj)$ corresponds to the ordered execution of events Ai and Bj. The application of $sequence(u, Ai, Bj)$ is defined as;

```
knowledge(Bj(u)) = sequence(Ai,Bj)|knowledge(u)
if Ai /in knowledge(u) and event-memory(u) = i.
```

This complex event is detected (its occurrence is added to KB) during the occurrence of event Bj, which can occur if in the previous state Ai had occurred, i.e. event-memory had stored i (and not if the memory is equal to null).

By using the observer event-memory and declaring which event had occurred before we can avoid the unintended semantics these operators can have, which are caused because the events, in the active database sense, are treated as if they occur at an atomic instant. This problem is discussed

in [45, 46] where also a solution is proposed by defining an interval-based effect semantics in terms of an interval-based event calculus formalization. The alternative interval-based semantics could be implemented in our framework by extending the definition of an event with the time of its occurrence and introducing the notions of event and time intervals.

The rest event operators (concurrent, aperiodic and periodic), which are used less often, cannot be straightforwardly expressed in our framework and an extension is required in order to include them as well.

3.2.4 A Supply Chain Management System

To demonstrate the expressiveness of our framework we applied it to an industrial case study that uses Event Condition Action rules to control the activities of its agents. These activities are inter-enterprise business processes and thus their verification is an important task. In [47] authors present an integrated work flow-supported supply chain management system that was developed so that Nanjing Jin Cheng Motorcycle Corporation in China and its suppliers could handle better their inner processes.

In the next figure a part of the inter-enterprise business process occurring in the supply chain is presented and more precisely the activities of the manufacturer agent. S and E are dummy activities that represent the start and end of the process, respectively. The arrowed edge represents the execution direction. The labels R_i $(i = 1, \ldots, 8)$ denote the corresponding ECA rules that are used to control the route of activities.

This business process consists of the following steps:

1. After the sales department accepts an order from a customer, a charge activity is initiated to wait for payment from the customer.

2. When the payment of the order is received and if it covers the total cost, the query inventory activity starts to check if the inventory level is enough to satisfy the order.

    (a) If it is enough, the deliver goods activity is initiated and the inventory level is adjusted by the inventory agent.

    (b) If it is not enough, the manufacture plan activity is activated that checks if there is enough material to produce goods.

        i. If there is enough material, a manufacturing plan is created by the product plan agent and then the manufacture activity starts.

        ii. If there is not enough material, a list of materials shortage is sent to the outsourcing agent, and an outsource activity is initiated. After the materials purchased from the supplier arrive, the manufacture activity is activated.

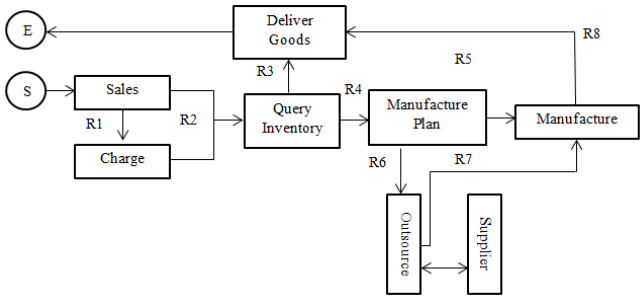


Figure 3: Representation of an inter-enterprise business process

3. When the products for the order have been produced, the deliver goods activity starts to ship goods to the customer.

4. The process is completed when the goods are delivered to the customer.

The inter-enterprise work flow model is based on a nested sub-processes model. In this, the originating process instance starts a sub process in a work flow enactment service and waits for a return message during its execution. The originating process chooses the sub-process according to its goal, requiring only the sub process to fulfill this goal and return results. With the support of this model, it is convenient to adjust the business process according to different requirements.

It can be easily adjusted by modifying some of the ECA rules and no code needs to be modified [47]. In this case, our framework due to its modularized structure can be used to verify the design of the adjusted model i.e. ensure that the system meets its new requirements, with minor changes in the initial specifications.

The proposed system consists of a set of business function agents whose tasks are to deal with outsourcing, production planning, sales, customer service, inventory, and so on. Each agent is an autonomic and independent entity. ECA rules are used to control the execution sequence of agents' activities. These rules are presented in the following table.

Rules $R1$-$R8$ were expressed in our framework according to the previous definitions. For example, the first rule was defined in CafeOBJ using the transitions endsales and stcharge. The first transition represents the event part of the rule and the second the action. The definition of the transition *endsales* can be seen below:

```
c-endsales : Sys department customer Nat -> Bool
c-endsales(S,Sales,C,N) = (order(S,Sales,C) = N) and
(event-memory(S) = null).

knowledge(endsales(S,Sales,C,N)) = (endsales|knowledge(S))
if c-endsales(S,Sales,C,N).
```

Table 4: ECA rules controlling the activities of the manufacturer

R1	On end(sales)	R5	On end(ManufacturePlan)
	Do st(charge)		if isMaterialsEnough
R2	On end(sales) and end(charge)		Do st(Manufacture)
	if payment $\geqslant$ totalprice	R6	On end(ManufacturePlan)
	Do st(QueryInventory)		if not isMaterialsEnough
R3	On end(queryinventory)		Do st(Outsource)
	if IsGoodsEnough	R7	On end(Outsource)
	Do st(DeliverGoods)		if ArrivedMaterials
R4	On end(queryinventory)		Do st(Manufacture)
	if not IsGoodsEnough	R8	On end(Manufacture)
	Do st(ManufacturePlan)		Do st(DeliverGoods)

```
event-memory(endsales(S,Sales,C,N)) = 1 if c-endsales(S,Sales,
C,N).
```

The effective condition *c-endsales* denotes that the event endsales can be detected when the sales department receives an order from a customer and if no other event had been detected in the previous state. The observer order returns the cost of the order a department receives from a customer. When the event is successfully detected its name enters the knowledge base and event-memory stores its identification number. The transition *stcharge* is defined as follows:

```
c-stcharge : Sys Nat customer -> Bool
c-stcharge(S,N,C1) = (endsales /in knowledge(S)) and
(event-memory(S) = 1).

event-memory(stcharge(S,N,C1)) = null if c-stcharge(S,N,C1).

knowledge(stcharge(S,N,C1)) = knowledge(S).

payment(stcharge(S,N,C1),C2) = pending if c-stcharge(S,N,C1)
and (C1 = C2).
```

The effective condition *c-stcharge* denotes that the action stcharge will occur if the event endsales belongs to the KB and event memory contains the id number of the event. After the execution of the action, the observer event-memory becomes null, knowledge base stays the same (because the occurrence of endsales event is needed for the detection of the complex event end(sales) *and* end(charge) of $R2$) and the payment of the customer is pending until a receipt is received. The sixth rule was defined in CafeOBJ using the transitions stoutsource and endmanufactureplan. The definition of the transition $endmanufactureplan$ is presented below;

```
c-endmanufactureplan : Sys bill inventory -> Bool
```

```
c-endmanufactureplan(S,B,I) = (materials(S,B,I) = computed)
and (event-memory(S) = null).

knowledge(endmanufactureplan(S,B,I)) = (endmanufactureplan|
knowledge(S)) if c-endmanufactureplan(S,B,I).

event-memory(endmanufactureplan(S,B,I)) = 5 if
c-endmanufactureplan (S,B,I).
```

The effective condition *c-endmanufactureplan* denotes that the event can be detected when it is computed if there are enough materials to produce goods for the order and if event-memory is null. When the event is detected the name of the event enters the knowledge base and the observer event-memory stores the number of the event, i.e. 5. The transition *stoutsource* is defined as follows;

```
-- stoutsource
c-stoutsource : Sys bill inventory agent -> Bool
c-stoutsource(S,B,I,A) = endmanufactureplan /in knowledge(S)
and (event-memory(S) = 5) and (materials(S,B,I) < enough) .

knowledge(stoutsource(S,B,I,A)) = (knowledge(S)/
endmanufactureplan) if c-stoutsource(S,B,I,A) .

event-memory(stoutsource(S,B,I,A)) = null if
c-stoutsource(S,B,I,A) .

list(stoutsource(S,B,I,A),A) = true if
c-stoutsource(S,B,I,A) .
```

The effective condition *c-stoutsource* declares that the action can be successfully applied if the event endmanufactureplan has been detected and the condition of the action holds, i.e. the materials are not enough. When the action occurs, the observer event-memory becomes null, the occurrence of the event is removed from the knowledge base and a list is sent to the outsourcing agent.

In a similar way we expressed all the rules in our framework. We also defined the transitions whose occurrence makes the detection conditions of the events true. For example, in order to detect the event endmanufacture, the products for the order must have been produced. Thus, we defined the transition *produceproducts*. When this transition is successfully applied, the value of the observer products becomes "produced", indicating that the event endmanufacture can be detected;

```
products(produceproducts(S)) = produced if c-produceproducts(S) .
```

In the above case study, the events may seem as simple propositional representations, or similar in format, but in the context of the whole specification they fully express the functionalities of the system. In order to specify this manufacturer agent, 18 transitions (12 that correspond to events and actions and 6 external transitions) and 14 observers were needed.

The most important feature of the proposed framework is the ability to verify the behavior of reactive rule-based intelligent agents using the proof score methodology, we have described in details in previous sections.

The type of properties that can be proved with the framework are safety properties, that hold in any reachable state of the system (called invariant properties), and liveness properties, which denote that something will eventually happen. For the supply chain system of the previous section, we proved that the process of delivering the goods to the customer must not be activated if the payment of the customer does not cover the total cost of the order. This is an invariant property, important for the purpose of the system. Invariant 1, is defined in CafeOBJ terms as;

```
inv1(S,C) = not(not(payment(S,C) >= cost(S,C)) and (delivered(S,
C) = true)).
```

At this point, we would like to mention some interesting points of the proof of this invariant. In the inductive step of invariant 1, CafeOBJ could not reduce the effective condition to either true or false for the transition *stdelivergoods*, and we had to split the cases as shown below;

```
open ISTEP
eq c-stdelivergoods(s,n1,i,c1) = false .
eq s' = stdelivergoods(s,n1,i,c1) .
red istep1 .
close

open ISTEP
eq c-stdelivergoods(s,n1,i,c1) = true .
eq s' = stdelivergoods(s,n1,i,c1) .
red istep1 .
close
```

Also the case denoted by the following equations returned false, for the transition *stcharge*. It was easy to understand that not all of the above equations could hold simultaneously in our system, and thus we used the invariant 2 to discard it (and CafeOBJ then returned true).

```
open ISTEP
-- eq c-stcharge(s,n1,c1) = true .
eq (endsales /in knowledge(s)) = true .
eq event-memory(s) = 1 .
eq c = c1 .
eq (payment(s,c1) >= cost(s,c1)) = true .
eq s' = stcharge(s,n1,c1) .
red inv2(s,c1) implies istep1 .
close

inv2(S,C) = not((event-memory(S) = 1) and
(payment(S,C) >= cost(S,C))).

inv3(S,C) = not((event-memory(S) = 3) and
not(payment(S,C) >= cost(S,C))).
```

Finally, invariant 3 was needed for the transition *stdelivergoods*. Following the proof scores method we successfully verified invariant 1 and the two lemmas that were needed to conclude the proof. The full specification and the proofs can be found at [48].

In order to compare the proposed framework with other similar approaches we first give a brief description of the related work. A lot of research concerning analysis of rule-based systems exists in the area of active databases. For example, in [49] authors present an overview of processing rules in production systems, deductive and active databases. A larger survey on the different approaches of reaction rules can be found in [50]. Most of the approaches addressing formal analysis of such systems however deal with checking properties such as termination, confluence and completeness. One such attempt to verify rule-based systems can be found in [51], where authors use Petri-nets to analyze various types of structure errors such as inconsistency, incompleteness, redundancy and circularity of rules. Also, ECA-LP [45] which is based on a labeled transaction logic semantics, supports state based knowledge updates including a test case based verification and validation for transactional updates.

Few papers targeting the verification of the behavior of active rule-based systems/agents exist. More precisely, in [52] authors describe a reasoning framework for Ambient Intelligence that uses the Event Calculus formalism for reasoning about actions and causality. Also, an approach to verify the behavior of Event-Condition-Action rules is presented in [40] where a tool that transforms such rules to timed automata is developed. Then the Uppaal tool is used to prove desired properties for a rule-based application. This last work is the closest to ours with the difference that in [40] authors use model checking, while our approach uses theorem proving techniques.

The proposed framework focuses on verifying the behavior of rule-based agents, rather than proving correctness properties or handling problems with negations, mainly for two reasons; first, the verification of such properties has been studied in many other approaches [51] and second the OTS/-CafeOBJ method does not study properties about the transitions of the system but analyzes their effects in the system's behavior. We believe that our framework has the following advantages over existing approaches; it can be used for the verification of complex systems due to the simplicity of the CafeOBJ language and its natural affinity for abstraction. Also, it has the ability to specify systems with infinite states (in contrast with approaches that use model-checking techniques) and it allows the re usability not only of the specification code but also of the proofs [26].

3.3 On Verifying Reactive Rules Using Rewriting Logic

In the previous sections we presented some first steps to use algebraic specification techniques in the area of reactive rules and more precisely, we proposed the use of OTS/CafeOBJ method to prove safety properties about reactive rule-based systems. We gave an Observational Transition System (OTS) semantics to Production and Event Condition Action rules so that verification of reactive rules can be supported. OTSs are described as equational theory specifications in CafeOBJ and the OTS/CafeOBJ method ([23, 24]) is then used to theorem prove that systems (formalized as OTSs) have desired properties. This approach has been effectively used for the specification of various complex systems [53] and the verification of invariant and liveness [54] properties of them.

The methodology proposed in the previous section and in [55] however cannot express naturally structure properties about reactive rules, such as confluence and termination. To this end, in this section we extend the previous approach by adopting a different logical formalism, so that the behavior of reactive rules can be formally analyzed in a seamless manner.

The extended framework can be used to; (1) formally specify reactive rules, (2) detect structure errors, like confluence and termination, and (3) prove invariant safety properties of the specified reactive rule based system, via theorem proving and model checking techniques. This diversity of options to verification (techniques and properties) offered by the resulted form of the reactive rules and the underlying logic, consists the main contribution of our work.

The properties of interest to rule systems verification include both safety properties and structure properties, such as confluence and termination of the rules. These properties are briefly described below;

A safety property is an assertion that a desirable property holds in all reachable states (i.e. is an invariant) of the rule-based system and is specific to the purpose of the specified application. Confluence concerns whether the result of executing a set of triggered rules depends on the execution order of the rules or not. A rule program is considered confluent in other words when from any initial state, all program executions lead to the same final state. Termination analysis aims to ensure that a set of rules will eventually terminate (i.e. reach a final state) and will not continue to trigger each other infinitely. A system may never terminate due to circular triggering of rules for example.

The behavior of rule based systems depends on their operational semantics. These are determined through the semantics of a rule execution procedure, which is usually called rule engine. A rule engine that executes production rules, consists of the following steps [56]:

1. Set the working memory to the initial state.

2. Build the set of all applicable and eligible rules. This set is called the agenda of the rule engine.

3. If the agenda is empty, the execution ends.

4. Otherwise, use a conflict set resolution strategy and choose a rule r in the agenda.

5. Update the working memory by executing the action of r. If the rule action contains several assignments, execute them in sequence.

6. Go to step 2.

The purpose of the rule eligibility strategy (step 2) is to avoid trivial infinite loops caused by applying again and again the same rule. It defines what a trivial loop is, and avoids them by making some rules ineligible. The purpose of the conflict set resolution strategy (step 3) is to pick the next rule to execute from the agenda. Again, several such strategies exist. Assigning a priority to each rule is a commonly used strategy.

Commercial engines employ such strategies to support logging execution traces, to provide simulation capabilities, and finally test and debug a rule set. For example, the problem of confluence can be solved by using priorities. It has been argued however that using this approach can be iterative since after prioritizing rules, say r1 and r2, a new pair of rules causing non-confluence may be identified [40]. The problem of termination can be solved by not allowing rules to trigger each other. However, this can reduce the usefulness of the language [40].

Even though most engines provide the support described above, they also present the discussed downsides and in addition they do not permit reasoning about the rule based system. We believe that formal methods can provide a feasible solution to this problem complementing the existing tools.

3.3.1 Reactive rules and CafeOBJ

We present here how reactive rules can be formally defined as a set of rewrite theory specifications in CafeOBJ. Recall that in a rewrite theory, states can be expressed as tuples of values $< a1, a2, b1, b2 >$ or as collections of observable values $(o1[p1] : a1)(o1[p2] : a2)(o2[p1] : b1)(o2[p2] : b2)$ (soups), where observable values are pairs of (parameterized) names and values.

The main difference between the two expressions is the following; when the states are expressed as tuples, the state expressions must be explicitly described on both sides of each transition. But when expressing states as soups, only the observable values that are involved in the transitions need to be described on both sides of each transition.

By adapting the definition presented in [55], here we define a set of reactive rules as rewrite theory specifications of OTSs expressed as collections

of observable values as follows;

Definition 13. A production rule is expressed as a term of the form R_i = *On* C_i *do* A_i where A_i can either denote a variable assignment, or an assertion, retraction, update of the knowledge base (add/remove/update facts from the KB respectively) or some other generic action with side effects. In the case where A_i denotes a variable assignment this is expressed by a transition rule of the form:

```
ctrans [Ri] (V: v0) D => (V: v1) D if  Ci = true.
```

Where, R_i is the label of the transition rule and v_0, v_1 are variables. Also, the keyword *ctrans* is used because the rule is conditional. The above rule states that the observable value V will become v_1 if the condition of the rule is true. Also, D denotes an arbitrary data type needed for the definition of the transition. When the result of action A_i is the assertion of the fact k_i to the knowledge base, its definition is the following;

```
ctrans [assert ki] (knowledge: K) D => (knowledge: (ki U K) D
if Ci /in K.
```

In the above rewrite rule knowledge is the observable value corresponding to the knowledge base and it is defined as a set of boolean elements, as before. When the result of action A_i is the retraction of the fact k_i from the knowledge base, its definition is;

```
ctrans [retract ki] (knowledge: K) D => (knowledge: K / ki) D
if Ci /in K.
```

When action A_i is an update action, its definition is the following;

```
ctrans [retract ki] (knowledge: K) => (knowledge: (K / ki) U kj)
if Ci /in K.
```

Finally, if A_i is a generic action and extra observable values (o_i) need to be used for its definition, we have;

```
ctrans [ai] (oi: vi) D => (oi: vj) D if Ci = true.
```

In order to express ECA rules as rewrite rules we use the definition below;

Definition 14. An Event Condition Action rule of the form $R_i := On\ E_i\ if\ C_i\ do\ A_i$ is defined in CafeOBJ terms as two transitions.

The first one specifies the event E_i and in particular the fact that the system after the detection of the event it stores its identification number in the observable value event-memory. This is defined as;

```
ctrans [Ei] (event-memory: null) => (event-memory: i) if c-ei
= true.
```

In the above rewrite rule, the value null of event-memory, denotes that no other event is detected at the pre state and *c-ei* is a boolean CafeOBJ term denoting the detection conditions for E_i.

The second transition rule specifies the action A_i. More precisely it defines that the system must respond to the detected event by performing the corresponding action, where A_i is again either a generic action or a predefined action of the rule language. The triggering of the action as a response to the event is simply defined by adding the condition that in the pre state the event memory will contain the index of the occurred event, i.e.

```
ctrans [ai] (event-memory: m) (oi: vi) => (event-memory: null)
(oi: vj) if  Ci = true and (m = i).
```

This ensures that only the guard of this transition rule will hold at the pre state and thus this will be the only applicable transition for that state of the system. Also after the occurrence of the action, event-memory will become null again denoting that the system is ready to detect another event.

Let us note here that in cases where the action of the rule may activate an internal event (say E_j), the observable value stores the id of the event, j, and becomes null again when the corresponding to the internal event action is applied (if it does not activate another internal event).

3.3.2 Running example.

We use as a running example a company's e-commerce web site [56] in order to explain better the definitions presented before. This company has customers with registered profiles on the site, which contain information about the customers' age and their category (Silver, Gold, or Platinum). When a customer puts items in his/her shopping cart a discount is computed based on the pricing policy of the company, i.e. on the customer's profile and the value of the cart. The behavior of this system is defined by the following three production rules;

1. The gold-discount rule implements a policy that increments the discount granted to Gold customers by 10 points, if their shopping cart is worth 2,000 or more.

2. The platinum-discount rule implements a policy that increments the discount granted to Platinum customers by 15 points, if their shopping cart is worth 1,000 or more.

3. The upgrade rule implements a policy that promotes Gold customers to the Platinum category, if they are aged 60 or more.

These rules can be written as a set of rewrite transition rules in CafeOBJ according to definition 13. First, the state of our system is formally described in the module below;

```
mod! State {
pr(Type + Nat)
[Obs < State]
```

```
-- configuration
op void : -> State {constr}
op _ _ : State State -> State {constr assoc comm id: void}
-- observable values
op category:_ : type -> Obs {constr}
op value:_ : Nat -> Obs {constr}
op age:_ : Nat -> Obs {constr}
op discount:_ : Nat -> Obs {constr} }
```

As we can see a state is defined as a set of the following observable values (*category* :)(*value* :)(*age* :)(*discount* :). The three last values are represented by natural numbers and for this reason the module imports the predefined module Nat. Also $pr(Type)$ imports a previously defined CafeOBJ theory which specifies the various customer types, i.e. gold, platinum and silver. Next, the rules $R1$-$R3$ are defined as a rewrite theory:

```
ctrans [gold] : (category: G) (value: V) (discount: M)
=> (category: G) (value: V) (discount: (M + 10))
if ((V >= 2000) and (G = gold)).

ctrans [platinum] : (category: G) (value: V) (discount: M)
=> (category: G) (value: V) (discount: (M + 15))
if ((V >= 1000) and (G = platinum)).

ctrans [upgrade] : (category: G) (age: N) (discount: M)
=> (category: platinum) (age: N) (discount: M)
if (G = gold) and (N >= 60).
```

The gold rewrite rule, states that if the observable value *category* is gold and the *value* is equal or greater than 2000 then the value *discount* will be increased by 10 points. The platinum rewrite rule, states that if the observable value *category* is platinum and the *value* is equal or greater than 1000 then the value *discount* will be increased by 15 points. The upgrade rewrite rule states that if the observable value *category* is gold and the *age* is 60 or more then the value *category* will become platinum.

As discussed in [56], there is an ambiguity between the upgrade and discount rule. If a gold customer is eligible to both being granted the gold discount and being upgraded to the platinum category, then this customer may end up with either a 15 or 25 per cent discount, depending on the execution order of the rules. This can be a hazard for the business application implementing this set of rules. We will present how such structural errors can be detected using our approach.

In particular, in this section we define some CafeOBJ operators which allow us to reason about confluence and termination properties of reactive rule based systems specified as rewrite logic theories. Also we demonstrate how existing operators can be used together with the proposed formalization of reactive rules to verify invariant properties about them.

3.3.3 Proving termination properties

Termination in a rule based system concerns with the existence of a state of that system where no more rules are applicable. More precisely;

Definition 15. A rule program's state s is terminating if and only if there is no infinite sequence $s \rightarrow s1 \rightarrow s2 \rightarrow ...$ In other words a state s is terminating if it leads to a state where no rules can be applied. That is, there exist two states such that; $s \rightarrow s'$ and $\neg(s = s')$ where s' is a final state. Based on this, we can check if a state terminates by defining the following predicate in CafeOBJ terms;

```
op terminates? : State -> Bool
terminates?(s) = s =(1,*)=>! (o1: v1) (o1: v2)
red terminates?(s) .
```

The expression $t1 = (1, *) => !t2$ indicates that the term matching to $t2$ should be a different term from $t1$ to which no transition rules are applicable. Also, the term $(o1 : v1)(o1 : v2)$ represents an arbitrary state and it depends on the observable values of the specified system. By reducing the above predicate, we ask CafeOBJ to find a *a final state* reachable from the state s. If *true* is returned (together with a final state) it means that the state s is terminating; if $false$ is returned it means that in the state s, no transition can be applied. Finally, the CafeOBJ reduction may not terminate, indicating that s is not terminating. Using the above predicate, we can check if the whole rule based system terminates or not, by defining the search to be performed for the initial state of the system;

```
op init : -> State
red terminates?(init).
```

When the number of reachable states reachable from init is small enough, the whole reachable state space can be checked by, $init = (1, *) =>$, where $*$ denotes infinity. Otherwise, the bounded reachable state space whose depth is d may be checked by, $init = (1, d) =>$.

Here we test the set of rules of the running example for termination. We must mention that in most real life applications the initial state of the system is explicitly defined during the design of the system. An e-shop site for example before the implementation could have the following characteristics; initially, no customer is registered at the site, when someone registers for the first time his/her category is silver, the discount is zero and so on.

It is possible however, for a system to be defined without explicitly defining its initial state. In such cases, we can still check the desired properties (confluence and termination) by defining an arbitrary initial state and then discriminate the cases based on the conditions of the transition rules[3]. Here

[3] In our example these cases are; (age $<$ 60 or age $>=$ 60), (discount = gold or discount

we present the most indicative cases of the running example;

```
-- case (a)
open RULES .
op s : -> State .
eq s = (category: gold) (value: 500) (age: 50) (discount: 0)
red terminates?(s) .
```

In this case CafeOBJ returns false and the following message, which is reasonable since no transition can be applied.

```
** No more possible transitions.
(false): Bool

-- case (b)
eq s = (category: gold) (value: 500) (age: 60) (discount: 0)
red terminates?(s) .
```

In this case where upgrade is the only applicable rule the CafeOBJ system returns true and the final state $(category : platinum)(value : 500)(age : 50)(discount : 0)$.

```
-- case (c)
eq s = (category: gold) (value: 2000) (age: 50) (discount: 0)
red terminates?(s) .
```

In this case the gold rule can be applied to s and to all reachable states from s. Thus in CafeOBJ the above reduction does not halt indicating that this initial state is not terminating. The same conclusion holds for the platinum rule as well.

Having detected this issue we can correct the rule base by adding constraints to the application of these rules, for example $(discount : (M1 + 10) <= 100)$ and $(discount : (M1 + 10) <= 100)$ respectively, since the discount cannot surpass this value. In this way the rules will stop triggering when the discount reaches the maximum value. When the same case is tested after adding the above constraints CafeOBJ finds the final state; $(category : gold)(value : 2000)(age : 50)(discount : 100)$.

3.3.4 Proving confluence properties

Once a rule based system has been checked for termination, it is important to be able to determine if it is confluent or not (if the rules do not terminate they will not be confluent either).

Definition 15. A rule program's state s is non-confluent if there exist two traces $trace_1$ and $trace_2$ from this state that lead to distinct states. That is, there exist two traces and three states such that; $s \xrightarrow{trace_1} s1$ and s $\xrightarrow{trace_2} s2$

= platinum), (value < 1000 or value >= 1000) and (value < 2000 or value >= 2000). For the last two only the following (value < 1000 or 1000 <= value < 2000 or value >= 2000) need to be checked.

and $\neg(s1 = s2)$, where $s1$ and $s2$ are final states. Based on this, we can check a state for non-confluence by defining the following predicate in CafeOBJ terms;

```
op notConfulent? : State -> Bool
notConfluent?(s) =(2,*)=>! (o1: v1) (o2: v2) .
red notConfluent?(s) .
```

The above reduction i.e. asks CafeOBJ to search if it can find starting from an arbitrary state s *two different final states* of the system. For this reason we use again the predicate with the exclamation mark at the end (final state) but in the number indicating the number of solutions we assign the value two (two different states). If two such solutions are found it means that the state s is not confluent. Otherwise if false is returned and one solution is found, the state is confluent. To check a rule based system for confluence we perform the search for the initial state of the system, as before, using the command:

```
red notConfluent?(init).
```

Here we test the set of rules of the running example for confluence. Again we can discriminate the cases for an arbitrary initial state. For example:

```
-- case (a)
open RULES .
op s : -> State .
eq s = (category: gold) (value: 500) (age: 60) (discount: 0)
red notConfluent?(s) .
```

In the above case where upgrade is the only applicable rule CafeOBJ returns false, as it finds one final state meaning that the state s is confluent. Now let us consider the state which is defined by the following observable values; the value of the items of the cart is equal to 2000 dollars, the age of the customer is 60 years old and her/his category is gold. This is the state we mentioned at the beginning of the section, in which the customer is eligible to both being granted the gold discount and being upgraded to the platinum category.

```
-- case (b)
eq s = (category: gold) (value: 2000) (age: 60) (discount: 0)
red notConfluent?(s) .
```

CafeOBJ returns true as it finds two solutions, denoting that s is not confluent as we expected. In particular it returns;

```
** Found [state 25] (category: platinum) (value: 2000) (age:
60) (discount: 90)
** Found [state 27] (category: platinum) (value: 2000) (age:
60) (discount: 95)
```

Using the command *show path id* we can see the two transition paths that cause the problem (and then we can add constraints in the conditions of the rules as before to solve this issue by letting for example the upgrade

rule to be applied first). Even though the presented example is quite simple it demonstrates that detecting such errors before the implementation of a rule based system can prove really helpful especially when designing complex critical systems.

3.3.5 Proving safety properties

The built-in CafeOBJ search predicate can also be used to prove safety properties for a system specified in rewriting logic (RWL). In this work, we are interested in invariant properties. For the verification of such properties model checking and/or theorem proving can be used; An invariant property can be model checked by searching if there is a state reachable from the initial state such that the desirable property does not hold [20]. This can be achieved using the following expression:

```
red init =(1,*)=>* p [suchThat c] .
```

In the above term c is a CafeOBJ term denoting the negation of the desired safety property. Thus, CafeOBJ will return true for this reduction if it discovers (within the given depth) a state which violates the safety property. This methodology is very effective for discovering (shallow) counterexamples. However, model checking does not constitute a formal proof and is complementary to theorem proving. Formal proofs are required when we are dealing with critical systems. In [20] a methodology to (theorem) prove safety properties of OTS specifications written in RWL is presented. This methodology can be used to reason about rule based systems expressed in our framework as we will demonstrate throughout the running example. For our rule based system an invariant safety property could be the following; *a customer cannot belong to the platinum category if his/her age is less than 60 years*. This is expressed in CafeOBJ terms as;

```
isSafe : State -> Bool .
isSafe((category: G) (value: V) (age: N) (discount: M)) = not
((G == platinum) and (N < 60)) .
```

The proof is done by induction on the number of transition rules of the system. First, the following operator is used [20];

```
vars pre con : Bool
check : Bool Bool -> Bool
check(pre, con) = if (pre implies con) == true then true
else false fi.
```

This operator takes as input a conjunction of lemmas and/or induction hypotheses and a formula to prove and returns true if the proof is successful and false if *pre implies con* does not reduce to true (this is why the built in == CafeOBJ operation is used, which is reduced to false iff the left and right hand side arguments are not reduced to the same term). Using this predicate the base case of the proof is successfully discharged using the following CafeOBJ code:

```
init = (category: gold) (value: 2000) (age: 50) (discount: 0).
red check(true, isSafe(init)).
```

The inductive step consists of checking whether from an arbitrary state, say s, we can reach in one step a state, say s', where the desired property does not hold. This can be verified using the following reduction [20]:

```
red s =(*, 1)=>+ s' suchThat (not check(isSafe(s),isSafe(s'))).
```

In the case where CafeOBJ returns false it means that it was unable to find a state s' such that the safety property holds in s and it does not hold i s'[4]. If a solution is found, i.e. the above term is reduced to true, then either the safety property is not preserved by the inductive step or we must provide additional input to the CafeOBJ machine. In the second case this input may be either in the form of extra equations defining case analysis or by asserting a lemma (in which case the new lemma has to be verified separately). Consider the inductive step where the *gold* transition rule is applied to s.

```
s = (category: gold) (value: 2000) (age: N) (discount: 0).
red s =(*, 1)=>+ s' suchThat (not check(isSafe(s),isSafe(s'))).
```

In the above equation *(category: gold) (value: 2000) (age: N) (discount: 0)* is an arbitrary state of the rule based system to which gold rule can be applied. CafeOBJ returns *false*, and thus the induction case is discharged. Consider the case where the *platinum* rule is applied;

```
s = (category: platinum) (value: 2000) (age: N) (discount: 0)
red s =(*, 1)=>+ s' suchThat (not check(isSafe(s),isSafe(s')))
```

CafeOBJ returns *false* for this case, thus the induction case is discharged. Following the same methodology the induction case for the upgrade rule was discharged as well, and thus the proof concludes. The full specification of the e-commerce site, the reasoning about the structure properties and proof of the invariant can be found at the [48].

In the area of analyzing the behavior of reactive rules, previous attempts, e.g. [57] and [58], propose the visualization of the execution of rules to study their behavior where rules can be shown in different levels of abstraction. More recent approaches related to the application of formal methods for analyzing rule based systems and relevant to ours, include the following; In [59] authors propose a constraint-based approach to the verification of rule programs. They present a simple rule language, describe how to express rule programs and verification properties into constraint satisfiability problems and discuss some challenges of verifying rule programs using a CP Solver that derive from the fact that the domains of the input variables are commonly very large. Finally, they present how to detect structure properties of a simple rule based system. In [56] authors analyze the behavior of

[4]To modularly verify each transition rule separately we usually, define for each such transition a new module which only contains one transition rule at a time.

Event Condition Action rules by translating them into an extended Petri net and verify termination and confluence properties of a light control system expressed in terms of ECA rules.

The proposed approach for the verification of rule programs is based on a different formalism; in particular it uses the OTS/CafeOBJ method and rewriting logic. To the best of our knowledge this is the first time it is used in the area of reactive rules. One motivation for this work was a recent advancement in the field, and in particular the methodology to theorem prove rewrite theories [20]. Compared to existing similar approaches, it has the following contributions.

Compared to [56] where structure errors are formally analyzed, our methodology can be used for the verification of both structure (confluence and termination) and safety properties for the specified rule system. This extends our previous work [55] where only safety properties could be proved. Second, when proving safety properties both model checking and theorem proving techniques can be applied, in contrast to [59] where only model checking support is provided. The combination of these two proving methods provides strong verification power. Model checking can be used to search the system for a state when the desired invariant property is violated (counter example) and next if no such state is discovered, theorem proving techniques can be applied to ensure that the system preserves the property in any reachable state. In this way infinite state systems can be specified. Also, CafeOBJ and Maude allow inductive data structures in state machines to be model checked and few model checkers exist with this feature. Finally our approach can be used for the specification and verification of complex systems due to the simplicity of the CafeOBJ language and its natural affinity for abstraction.

The proposed methodology does not come without limitations. One possible limitation could be the fact that researchers should be familiar with the CafeOBJ formalism in order to use the proposed approach. However, we believe that the mapping from reactive to rewrite rules is natural enough and the verification method has a clear structure, thus allowing non-expert users to adopt our methodology with minimum effort.

3.4 Formal Analysis and Verification Support for Reactive rule-based Agents

In this section we compare the methodologies presented in previous sections, for the specification and verification of intelligent systems whose behavior is expressed in terms of reactive rules, and report on some lessons learned after applying them in several case studies. We present, through a case study, an expanded framework which expresses the functionality of Production and Event Condition Action rules in terms of equational and rewrite transition rules, written in CafeOBJ. The proposed methodology, except from supporting reasoning about the specified rule-based system, verification of safety properties of the rules and detection of termination and confluence errors of the rules, it also provides a clear understanding of the specified rule based system by simulating the execution behavior of the rules. We also demonstrate a tool that translates a set of reactive rules into CafeOBJ rewrite rules, thus making the verification of reactive rules possible for inexperienced users. The two last points are the main contribution of the presented work.

We recall the two basic reactive rule families: Production rules and Event-Condition-Action rules, and we emphasize on the definition of *events*.

A Production rule is a statement of rule programming logic, which specifies the execution of an action in case its conditions are satisfied, while Event Condition Action (ECA) rules define an explicit event part which is separated from the conditions and actions of the rule.

The events of ECA rules can be combinations of atomic events activated by environmental or internal changes and based on that, they are usually classified as external and internal. These changes are captured by environmental and local variables, respectively. More precisely, external events are produced by sensors monitoring environment variables [56]. This means that environmental variables are used to represent environment states that can be measured by sensors but not directly modified by the system. In this way, environmental variables capture the nondeterminism introduced by the environment. Instead, local variables can be both read and written by the system. An external event can be activated when the value of an environmental variable crosses a threshold; on the other hand, internal events can only be activated by the actions of ECA rules. Internal events are useful to express internal changes or required actions within the system. These two types of events cannot be mixed within a single ECA rule. Thus, rules are external or internal, respectively. The condition part of an ECA rule is a boolean expression on the value of environmental and local variables. The last part of a rule specifies which actions must be performed. Most actions are operations on local variables which do not directly affect environmental variables. Thus, environmental variables are read-only from the perspective of an action. Also, actions can activate internal events. Finally, to han-

dle complex action operations, the execution semantics can be sequential or parallel.

We should mention that while we had first thought that it would be better to give formal semantics to a specific Rule Markup language (e.g. Reaction RuleML [41]) we chose to formalize reactive rules expressed in a more generic style, because in this way more languages can be covered. Besides, in most cases the semantics of the rules are independent of the syntax of the specific language. Also, many case studies and related work in the literature express the rules in this way, so it is easier to test the proposed methodologies.

3.4.1 A light-control intelligent system

We describe the proposed framework[5] through a case study that uses 10 ECA rules to define the behavior of a smart system. In the above table, rules $r1$ to $r10$ are presented, taken from [56]. These rules specify a light-control intelligent system which attempts to reduce energy consumption by turning off the lights in unoccupied rooms or in rooms where the occupant is asleep, using sensors. The system also provides automatic adjustment for indoor light intensity based on the outdoor light intensity.

The values measured by the sensors are stored in environmental variables. The measure of a motion sensor that detects whether the room is occupied or not is expressed by the boolean variable *Mtn*. A pressure sensor detects whether the person is asleep and this information is stored in the boolean environmental variable *Slp*. A light sensor, whose measure is expressed by the variable *ExtLgt* ($\in 1, \ldots, 10$), is used for monitoring the outdoor lighting.

MtnOn, MtnOff and *ExtLgtLow* are external events activated when the environmental values cross a threshold. *MtnOn* and *MtnOff* occur when variable *Mtn* changes from false to true or from true to false, respectively. *ExtLgtLow* occurs when variable *ExtLgt* drops below 6. Rule *r1* initializes the local variable *lgtsTmr* to 1 whenever the motion sensor detects no motion and the lights are on.

Internal events model internal system actions. For example, internal event *SecElp* models the system clock and is activated when the variable *lgtsTmr* has the value 1. The timer then increases as minute elapses, provided that no motion is detected (rule $r2$). If the timer reaches 6, internal event *LgtsOff* is activated to turn off the lights and to reset *lgtsTmr* to 0 (rule $r3$). Internal event *LgtsOff*, activated by rule $r3$ or $r7$, turns the lights off and activates another check on outdoor light intensity through internal event *ChkExtLgt* (rule $r4$). *ChkExtLgt* activates *LgtsOn* if *ExtLgt* drops below 6 (rule $r5$). Internal event *ChkMtn*, activated by rule $r6$, activates

[5]For the revision in the definitions and the additions in the proposed framework we refer the reader to [60].

Table 5: ECA rules controlling the lights intensity of a house

(R1)	When the room is unoccupied for 6 minutes, turn off lights if they are on.
r1	on MtnOff if (intLgts $>$ 0 and lgtsTmr = 0) do set (lgtsTmr, 1) par activate(SecElp)
r2	on SecElp if (lgtsTmr $\geqslant$ 1 and lgtsTmr $<$ 6 and lMtn = 0) do increase (lgtsTmr, 1)
r3	on SecElp if (lgtsTmr = 6 and lMtn = 0) do (set (lgtsTmr, 0) par activate (LgtsOff))
r4	on LgtsOff do (set (intLgts, 0) par activate (ChkExtLgt))
(R2)	When lights are off, if external light intensity is below 6, turn on lights.
r5	on ChkExtLgt if (intLgts = 0 and lExtLgt $\leqslant$ 5) do activate (LgtsOn)
(R3)	When lights are on, if the room is empty or a person is asleep, turn off lights.
r6	on LgtsOn do (set (intLgts, 6) seq activate (ChkMtn))
r7	on ChkMtn if (Slp = 1 or (Mtn = 0 and intLgts >= 1)) do activate (LgtsOff)
(R4)	If the external light intensity drops below 5, check if the person is asleep and set the lights intensity to 6. If the person is asleep, turn off the lights.
r8	on ExtLgtLow if (lSlp = 0) do set (intLgts, 6)
r9	on ExtLgtLow if (lSlp = 1) do set (intLgts, 0)
(R5)	If the room is occupied, set the lights intensity to 4.
r10	on MtnOn do (set (intLgts, 4) par set (lgtsTmr, 0))

LgtsOff if the room is unoccupied and all lights are on, or if the room is occupied but the occupant is asleep (rule $r7$).

Formal analysis using Equational Logic. First, to specify this system as an equational transition system using our definitions, we will need the following observers. The five first observers are used in order to observe the variables' (local and environmental) changes. The last two are used to model the detection of the events.

```
-- observers
Mtn : State -> Bool
```

```
ExtLgt : State -> Nat
Slp : State -> Bool
lgtsTmr : State -> Nat
intLgts : State -> Nat
event-memory : State -> Name
Mtn-memory : State -> SetofNames

-- variables and constants
var S : State
ops MtnOn, MtnOff, ExtLgtLow, LgtsOff, ChkExtLgt, ChkSlp, ChkMtn
: -> Name
```

Where, *State* is the hidden sort denoting the state of the system and S a variable of the same sort. The events of the rules (internal and external) are declared as constants of the sort $Name$. The transitions of the system (and also constructors of State) are the actions and the external events. Their definition can be seen below:

```
E1, E2, E8, E10 : State -> State
A1, A2, A3, A4, A5, A6a, A6b, A7, A8, A9, A10 : State -> State
```

The event of rule 8 is then described by the following conditional equations;

```
event-memory(E8(S)) = ExtLgtLow if ExtLgt(S) <= 5 and
event-memory(S) = null
```

The event *ExtLgtLow* is detected whenever the external lights' intensity drops below 6 and if no other event has been detected in the previous state. The action of the rule is defined as follows;

```
intLgts(A8(S)) = 6 if Slp(S) = false and event-memory(S)
= ExtLgtLow
```

Internal lights intensity will be set to 6, if the occupant is not asleep and the event *ExtLgtLow* has been detected in the previous state. The event of rule 1 is defined as follows;

```
Mtn-memory(E1(S)) = (Mtn-memory(S )|MtnOff)/MtnOn  if not
(MtnOff /in Mtn-memory(S))

event-memory(E1(S)) = MtnOff if event-memory(S) = null and
Mtn(S) = false and not (MtnOff /in Mtnmemory(S))
```

In this rule, the extra observer *Mtn-memory* is used because the external event *MtnOff* should only be re-detected if the event *MtnOn* occurs first, and vice versa (specific detection order of events). For this reason with the first equation we define that *MtnOff* will occur (and will be stored in the observer *Mtn-memory*) if it does not belong in the observer *Mtn-memory* (in other words if it had not occur before). Also, the only way to be removed from the observer *Mtn-memory* is by the occurrence of the event *MtnOn*. The same constrain holds for the event *MtnOn*. This ensures that *MtnOff* (resp. *MtnOn*) cannot be perpetually detected and cause the system to

loop. The second equation denotes that the event *MtnOff* is detected and its name is stored in the observer event-memory if there is no motion in the room, if the event *MtnOff* had not occurred before and if no other event has been detected in the previous state. The action of the rule is defined as;

```
event-memory(A1(S)) = SecElp if lgtsTmr(S) = 0 and intLgts(S)
> 0 and event-memory(S) = MtnOff

lgtsTmr(A1(S)) = 1 if lgtsTmr(S) = 0 and intLgts(S) > 0 and
event-memory(S) = MtnOff
```

After the detection of the event *MtnOff*, and if lights timer in the previous state was zero and the internal lights were on (greater than zero), lights timer will take the value 1 as the result of the action of the rule. Also the internal event *SecElp* will be activated. Rule 3, for example, is triggered by the action of the previous rule ($r2$) and thus we do not have to specify its event. The equational transition rule $A3$ describes the action of the rule, as follows:

```
lgtsTmr(A3(S)) = 0 if  ((Mtn(S) = false) and
(lgtsTmr(S) = 6) and (event-memory(S) = SecElp)) .

event-memory(A3(S)) = LgtsOff if  ((Mtn(S) = false) and
(lgtsTmr(S) = 6) and (event-memory(S) = SecElp)) .
```

The timer (observer *lgtsTmr*) will be set to zero, if the effective condition of the rule holds. This means that in the previous state the sensors had not detect any motion in the room, the timer had the value 6, and the event *SecElp* had been successfully detected. At the same time the internal event *LgtsOff* will be activated and stored in the observer event-memory.

In a similar way the rest of the rules are defined in our framework. For the verification of the system, when it is expressed in equational logic, we use Cafe OBJ's theorem proving technique [25] to verify desired safety properties of the rules. For our rule based system an invariant safety property could be the following; *the lights cannot be turned off if someone is in the room and he/she does not sleep.* This is defined in CafeOBJ terms as follows:

```
inv1(S) = not((intLgts(S) = 0) and (Mtn(S) = true) and
(Slp(S) = false)).
```

Suppose now we have the following initial state;

```
event-memory(init) = null
Slp(init) = true
Mtn(init) = true
Mtn-memory(init) = null
intLgts(init) = 1
lgtsTmr(init) = 0
ExtLgt(init) = 4
```

If we use the reduction *red inv1(init)*, CafeOBJ returns *true* for this initial state. For proving that invariant 1 holds when the event and the action of rule 8 are applied we used the following proof passages:

```
open ISTEP
s' = E8(s) .
red istep1 .
close

open ISTEP
s' = A8(s) .
red istep1 .
close
```

In both cases CafeOBJ returned true and thus the inductive step for rule 8 was discharged. The full specification of the light-control system as an equational transition system, and the proof of the invariant for the rest of the rules can be found at [48].

Formal analysis using Rewriting Logic. A state in rewriting logic is described as a collection (soups) of observable values, as we have already mentioned. In order to specify the light-control system using our second methodology, we use the following observable values;

```
event-memory:_ : Name -> Obs
Mtn-memory:_ : SetofNames -> Obs
intLgts:_ : Nat -> Obs
Mtn:_ : Nat -> Obs
lgtsTmr:_ : Nat -> Obs
Slp:_ : Nat -> Obs
ExtLgt:_ : Nat -> Obs
```

An arbitrary state of the system is defined as; *(event-memory: e) (Slp: s)(Mtn: l)(Mtn-memory: m)(intLgts: i)(lgtsTmr: t)(ExtLgt: x)*, where *e, s, l, m, i, t, x* are predefined variables which denote arbitrary values for the corresponding sorts. Rule $r6$ for example, which denotes that when the event *LgtsOn* is detected the lights intensity is set to 6 and in sequence an internal event that checks if there is motion in the room is activated, can now be defined as two rewrite transition rules in CafeOBJ terms, as follows:

```
trans [A6a] (event-memory: LgtsOn) (Slp: s) (Mtn: l)
(Mtn-memory: m)(intLgts: i) (lgtsTmr: t) (ExtLgt: x)
=> (event-memory: LgtsOn) (Slp: s) (Mtn: l)
(Mtn-memory: m)(intLgts: 6) (lgtsTmr: t) (ExtLgt: x).

ctrans [A6b] (event-memory: LgtsOn) (Slp: s) (Mtn: l)
(Mtn-memory: m)(intLgts: i) (lgtsTmr: t) (ExtLgt: x)
=> (event-memory: ChkMtn) (Slp: s) (Mtn: l) (Mtn-memory: m)
(intLgts: i) (lgtsTmr: t) (ExtLgt: x) if (i = 6).
```

To denote this rule we do not need to define an event transition, since it is activated by the internal event of the previous rule ($r5$). The first rewrite transition ($A6a$) specifies the first part of the action, i.e. sets internal lights intensity to 6, while the second conditional transition will be applied if in the previous state transition $A6a$ had been effectively applied and it will activate the internal event *ChkMtn*.

Rule $r8$, which denotes that if the external light intensity drops below 5 and the person is asleep, lights intensity is set to 6, is defined in two steps as follows;

```
ctrans [E8] : (event-memory: null) (Slp: s)
(Mtn: l) (intLgts: i) (lgtsTmr: t) (ExtLgt: x)
=> (event-memory: ExtLgtLow) (Slp: s)
(Mtn: l) (intLgts: i) (lgtsTmr: t) (ExtLgt: x)
if (x <= 5) .
```

Event *ExtLgtLow* is successfully detected and stored in the observable value event-memory when the sensor detects that external light density drops below 5 and if no other event has been detected in the previous state. The definition of the action of the rule can be seen below;

```
ctrans [A8] : (event-memory: ExtLgtLow) (Slp: s)
(Mtn: l) (intLgts: i) (lgtsTmr: t) (ExtLgt: x)
=> (event-memory: null) (Slp: s) (Mtn: l)
(intLgts: 6) (lgtsTmr: t) (ExtLgt: x)
if (s = false).
```

The action of the rule sets the internal lights intensity to 6 as a reaction to the detected event, if the person is asleep. Also, event-memory becomes null. Rule $r3$, which denotes that when the event *SecElp* is detected and if the room is unoccupied for 6 minutes, the timer is set to zero and in parallel an internal event that turns off the lights is activated, can now be defined as a rewrite transition rule in CafeOBJ terms, as follows:

```
ctrans [A3] : (event-memory: SecElp) (Slp: s)
(Mtn: l) (intLgts: i) (lgtsTmr: t) (ExtLgt: x)
=> (event-memory: LgtsOff) (Slp: s) (Mtn: l)
(intLgts: i) (lgtsTmr: 0) (lExtLgt: x)
if (l = false) and (t = 6).
```

To denote this rule we do not need to define an event transition, since it is activated by the internal event of the previous rule (r2). The rewrite transition ($A3$) specifies the action of the rule, i.e. sets the timer to zero and also activates the internal event *LgtsOff*, if the condition of the rule holds.

In a similar way we define the rest rules of the system. The full specification of the light-control system as a rewrite transition system can be found at [48]. For the verification of the system, this methodology allows us to verify both the structure and the behavior of specified rule-based system. In respect to the structural properties; with the help of CafeOBJ's search engine and the operators we have defined [61], we can *(a)* detect termination and *(b)* confluence errors for the system, and *(c)* simulate the execution behavior of the rules.

In order to check a rule based system for termination we must perform the search for all initial states of the system, using the command:

```
red terminates?(init).
```

Recall that if *true* is returned and a *final state*, it means that this state of the system will terminate. If *false* is returned it means that in this initial state, no transition can be applied or that the reachable state(s) from this state is(are) not final. Finally, the rewriting may *not terminate* because CafeOBJ's rewriting system may apply a transition rule on and on.

In the last case there is no reason to check the system for confluence since if a state is not terminating it is not confluent either. In the first two cases, we proceed with checking the state for confluence errors. Suppose that the initial state[6] of the system is the following; The lights are off, the room is empty (no one is sleeping), the intensity of the external lights is low, and we wish to see if this system will terminate or not.

```
set trace on
init = (event-memory: null) (Slp: 0) (Mtn: 0)
(Mtn-memory: null)(intLgts: 0) (lgtsTmr: 0) (ExtLgt: 4)
s' = (event-memory: e1) (Slp: s1) (Mtn: l1)
(Mtn-memory: m1) (intLgts: i1) (lgtsTmr: t1) (ExtLgt: x1)
red terminates?(init)
```

The command *set trace on*, instructs CafeOBJ to show in details, the performed rewrites. CafeOBJ for the above reduction returns the result: *true* and *found state 3*. This means that, state 3 is a final state reachable from the initial state of the system.

With the command *show path id*, we can see the applied transitions that lead to this state and their order. In our case, CafeOBJ returns the sequence of the transitions *E8* and *A8*. This result is as expected because according to the rules and our initial state, the external event *ExtLgtLow* will be activated (as the external light intensity is low) and since no one sleeps, the internal lights will be set to 6. This is a final state of the system as no other transitions can be applied. Having checked the system for termination, we can proceed with checking for its confluence.

In order to check a rule based system for non-confluence we perform the search for all the initial states of the system. Recall that if CafeOBJ finds *two solutions* it means that the state is not confluent. However, if *true* is returned together with the statement *No more possible transitions*, we must continue the analysis of the rule based system by conducting one more test to verify the confluence of the rules. *This last check basically simulates the execution behavior of the rules.*

To clarify why this final check is required, suppose we have the following initial state; The lights are on, the room is occupied but no one is sleeping and the intensity of the external lights is low (below 5). This is expressed as; If we check this initial state for confluence CafeOBJ will return the result; *found state 3* and *no more possible transitions*. In that case, it is not clear

[6]As we have already mentioned, in cases where the initial state of the system is not known beforehand we can define the set of possible initial states by defining an arbitrary initial state and discriminating the cases based on the conditions of the transition rules.

how the initial state behaves and further analysis is required. For this reason we use the following command, which returns *all the reachable states* from the initial state of the system.

```
red init =(*,*)=>* S
```

This, in combination with the command *show path id*, can be used in order to *simulate the execution behavior of the rules* in our framework. In our running case study, if we apply the above reduction in combination with the command *show path id* we get the following result:

```
found state 0 (init)
found state 1 (init - E8)
found state 2 (init -  E10)
found state 3 (init - E8 - A8)
found state 4 (init - E10 - A10)
found state 5 (init - E10 - A10 - E8)
```

This result is graphically presented in the following figure.

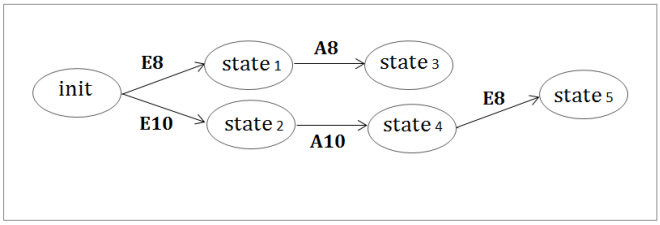


Figure 4: Graphical representation of the reachable states from the initial state of the system

In this way we get a clear perception of the execution of the rules. Now, due to the fact that CafeOBJ system visits each reachable state once, using this check we can discover that a state can be either:

- final, i.e. no transition can be applied in it, or
- in between, i.e. a transition can be applied in it and will lead to an already visited final state, or finally
- part of a loop, i.e. a transition can be applied in it but it will lead to a non-final state already visited in the same state path

In our case study, state 3 is final as the first check already showed. State 5 is an in between state because if transition $A8$ will be applied in it, it will lead to state 3 again (that is why CafeOBJ does not apply $A8$). So this initial state behaves well as it leads to the same final states.

Discovering a loop using the proposed methodology allow us to isolate the rule that is responsible for it as well as the condition which causes it.

This useful information can be used in order to change the rules and redesign the system.

Regarding the behavior of the rules; the built-in CafeOBJ search predicate can also be used to prove safety properties for a system specified in rewriting logic, as we have already mentioned. Proving invariant properties using model checking can be achieved by searching if there is a state reachable from the initial state such that the desirable property does not hold [20].

For example, consider an initial state in which the room is occupied, the person sleeps, the lights are on and the intensity of the external lights is low;

```
init = (event-memory: null) (Slp: true) (Mtn: true) (intLgts:
true) (Mtn-memory: null) (lgtTmr: 0) (lExtLgt: 4).

red init =(1,*)=>* (event-memory: e1) (Slp: false) lMtn: true)
(intLgts: false) (Mtn-memory: m1) (lgtsTmr: t1) (ExtLgt: x1).
```

CafeOBJ returns *false* for the above reduction, as it cannot find a state reachable from this initial state, in which the lights are off, the room is occupied and the person does not sleep. To obtain a formal proof about the desired property, we use the methodology presented in [20] that allows us to theorem prove safety properties of specifications written in rewriting logic.

Suppose we are interested in the same invariant property; *the lights cannot be turned off if someone is in the room and he/she does not sleep*. This can be expressed in CafeOBJ using the rewriting approach as follows;

```
isSafe : State -> Bool
isSafe((event-memory: e1) (Slp: s1) (Mtn: l1) (intLgts: i1)
(Mtn-memory: m1)(lgtsTmr: t1) (ExtLgt: x1)) = not ((i1 == 0)
and (l1 == true) and (s1 == false))
```

Suppose also we have the following initial state:

```
(event-memory: null) (Slp: true) (Mtn: true) (intLgts: 1)
(lgtsTmr: 0) (ExtLgt: 4) (Mtn-memory: null).
```

For the base case, all we have to do is to check if the following term reduces to *true*, which it does for this initial state.

```
red check(true, isSafe(init)).
```

Recall that the inductive step consists of checking whether from an arbitrary state, say s, we can reach in one step a state, say s', where the desired property does not hold. Consider the inductive step where for example, the transition rule $E8$ is applied to s.

```
-- transition E8
eq s = (event-memory: null) (Slp: s1)
(Mtn: l1) (intLgts : i1)(Mtn-memory: m1)
(lgtsTmr: t1) (ExtLgt: 4) .
red s =(*,1)=>+ s' suchThat (not check(isSafe(s),isSafe(s'))) .
```

In the code above *(event-memory: null) (Slp: s1) (Mtn: l1) (intLgts : i1)(Mtn-memory: m1) (lgtsTmr: t1) (ExtLgt: 4)* is an arbitrary state of the rule based system, in which rule $E8$ can be applied. CafeOBJ returns false, and thus this induction case is discharged. Consider now the inductive step where the transition rule $A8$ is applied to s.

```
-- transition A8
eq s = (event-memory: ExtLgtLow) (Slp: false)
(Mtn: l1) (intLgts : i1)(Mtn-memory: m1)
(lgtsTmr: t1) (ExtLgt: x1) .
red s =(*,1)=>+ s' suchThat (not check(isSafe
(s),isSafe(s'))) .
```

Where again, *(event-memory: ExtLgtLow) (Slp: false) (Mtn: l1) (intLgts : i1) (Mtn-memory: m1) (lgtsTmr: t1) (ExtLgt: x1)* is an arbitrary state of the rule based system, in which rule $A8$ can be applied. CafeOBJ returns false, and thus the induction step for rule 8 is discharged.

In the same way, the induction case for all the transition rules are discharged, and thus the proof concludes. The whole specification of the rule based system and the proof can be found in [48].

3.4.2 From reactive rules to CafeOBJ rewrite rules

The developed tool is written in Java and takes as input a set of reactive rules, written using the generic style described before, and automatically produces a set of rewrite transition rules written in CafeOBJ, implementing the definitions of the rewriting methodology.

The reactive rules specification has to obey in some syntactic guidelines which are described in the tool. For demonstration purposes, we present through the following screenshot, part of the transformation of the set of eca rules of the running case study.

The developed tool hides the details of the translation. Moreover, except from the automatically generated rewrite rules (figure 5, module RULES), it also creates the rest of the CafeOBJ specification which is required so that the output specification can be given as input to the CafeOBJ processor. In this way there is no need for the user to add any additional information, like module and sort declarations, observers, variables and so on (figure 5, module STATE). After the transformation, the user can use CafeOBJ directly in order to further analyze and verify the behavior of the rules.

We present here some lessons learned throughout our research in formalizing reactive rule based systems. A more general comment is that we believe that equational and rewriting logic is easier to learn than other logics, such as higher order for example, because they are similar to everyday life reasoning. Hence, we believe that the verification with CafeOBJ is easier to learn than those with other methods.

Also, transition rules expressed either in equational or in rewriting logic

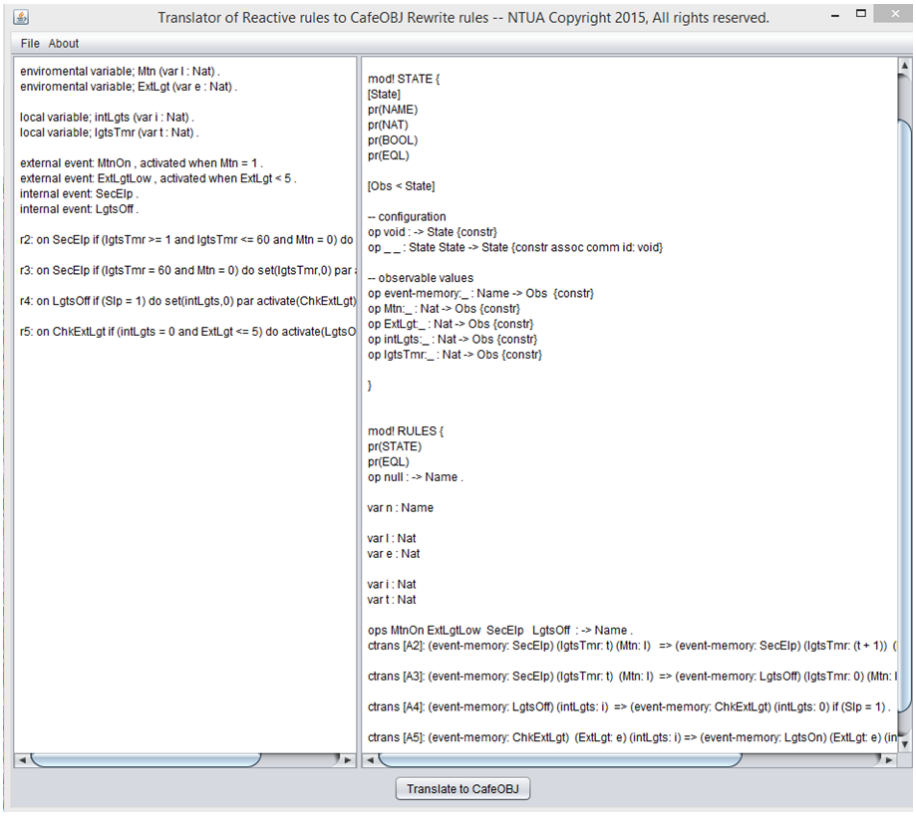


Figure 5: Output of the tool - generated rewrite rules and executable CafeOBJ specification

are closer to reactive rules researchers mindset (since their reasoning is similar to that of rules), thus the transformation is more straight forward. But that was not obvious from the beginning.

One of the difficulties we faced during our research in reactive rules, was the semantic difference between events and actions, i.e. the fact that while events *can* occur at anytime and can be straightforwardly mapped to transitions, actions *must* be executed after the detection of their triggering events. This issue was addressed by mapping an ECA rule into two different transition rules and by introducing appropriate observers that can handle this difference (by helping us decide if the system must react to a transition or treat it as an incoming event).

Another difficult point was selecting the most appropriate definition of termination and confluence for rule-based systems that can also be expressed in Cafe OBJ terms. To overcome this, we conducted a thorough search in the related literature and more precisely we based our definitions in those of [59], which had the level of abstraction that met our purposes.

As to which of the proposed methodologies is better, we should mention

that while the equational approach has better modeling capabilities as it provides succinct, composable specifications and provides stronger verification support, in order to decide which of the two is the most suitable, we have to take into account that these frameworks aim in bringing reactive rules' researchers closer to the area of formal methods. Thus we have to mainly considerate their needs and focus. To this end, we believe that the equational approach should be adopted when the specified system is complex and critical and we need a really expressive formalism for modeling the reactive rule-based system. However, the rewriting approach is better suited for the reactive rules researchers community, as it is more natural and easier to use. Also the rewriting logic approach offers a seamless framework for verifying reactive rules as both safety properties and structure errors can be checked. Finally it supports both theorem proving and model checking techniques. Thus we believe that in most cases the rewriting approach is preferred.

For this reason and in order to make the rewriting approach even more friendly for the potential users, we have developed a tool that automatically transforms a set of reactive rules into a set of rewrite rules in CafeOBJ.

Sometimes though, it is difficult for inexperienced users to do interactive theorem proving especially for complex systems. This is another reason why we believe that the rewriting logic approach is better suited for verifying reactive rule-based systems since it supports model checking as well, which is easier to learn and use. Also we hope that through this work which presents how to verify reactive rules with simple steps, the verification process will become clear even for inexperienced users.

4 Context-aware Adaptive Systems

"Systems that can anticipate the needs of users and act in advance by 'understanding' their context" [Chen 2003]

"Systems that are able to adapt their operations to the current context without explicit user intervention. Context-awareness here aims at increasing usability and effectiveness by taking environmental context into account" [Dustdar 2006]

Nowadays computers are becoming more pervasive in every aspect of our lives. It no longer makes sense for systems to work independently; they need to cooperate with each other, with the context and with the user. We see computers and microprocessors being used in many items that make our lives more convenient. Even the simple alarm clock or microwave has microprocessors inside.

"Imagine John, who checks in a hotel and has some preferences, say for example he would like to wake up at 6am, would like the AC to operate at his preferred temperature and humidity, would like his breakfast served at 8am. Usually John would have to take the effort to set up his preference manually and waste time. With the help of communicating devices and the use of his context, these preferences can be communicated to the devices in the hotel room automatically. According to his context his alarm clock would ring at 6am, his AC temperature adjusted to suit his requirement and his breakfast delivered at 8am etc. Now if another person checks into the same room after John has moved out, all the above mentioned utilities would operate differently as per the guest's preferences. The same room, under different contexts behaves differently" [62].

Context should be created spontaneously for any kind of task be it travel booking or purchasing electronic items. User wants the results to suit best his needs. As computers and electronic devices become ubiquitous, the need for an automatic method of adjusting their behavior to our needs arises.

Pervasive computing envisions a seamless and distraction-free environment of distributed and heterogeneous applications and devices that utilize resources in their environment. Devices and applications are context-aware, meaning that they can sense changes to their executing environment and manage information automatically and transparently. Recent technological advances in mobile devices as well as wireless and sensor networks make it possible to construct pragmatic, large-scale applications for pervasive computing. Large-scale applications have the potential to span different infrastructures (wireless networks, sensors, services), combine different technologies (e.g., communications, middleware) and device types, to offer an

integrated pervasive framework with rich capabilities across many conceptual application layers.

SatNav as context-aware system. In a Satellite Navigation System (SatNav), the current location is the primary contextual parameter that is used to automatically adjust the visualization (e.g. map, arrows, directions, etc.) to the user's current location. However, looking at current commercial systems, much more context information is used and much of visualization has been changed. In addition to the current GPS position, contextual parameters may include the time of day, light conditions, the traffic situation on the calculated route or the user's preferred places. Beyond the visualization and whether or not to switch on the backlight, the calculated route can be influenced by context, e.g. to avoid potentially busy streets at that time of day [63].

Automatic light as context-aware system. At house entrances and in hotel hallways automatic lights have become common. These systems can also be seen as simple context-aware systems. The contextual parameters taken into account are the current light conditions and if there is motion in the vicinity. The adaptation mechanism is fairly simple. If the situation detected is that it is dark and that there is someone moving, the light will be switched on. The light will then be on as long as the person moves, and after a period where no motion is detected, the light will switch off again. Similarly, the light will switch off if it is not dark anymore.

The GUIDE project. The GUIDE project (GUIDE 2001) at Lancaster University was the first larger and public installation of a research prototype to explore context-awareness in the domain of tourism. It focused on how context can be used to advance a mobile information system for visitors to the historic town of Lancaster.

MobileWARD - Mobile Electronic Patient Record, Aalborg University, Denmark. MobileWARD is a prototype designed to support morning procedure tasks in a hospital ward, and is able to display patients lists and patient information. The device presents information and functionality according to the location of the nurse and the time of the day. This project simulates the context events linked to the location of the staff member. Patients are chosen through a patient-list or an activation of a barcode at the bed-side. The scale and complexity of such application types pose new challenges. Adaptive programs are generally more difficult to specify, verify, and validate due to their high complexity. Particularly, when involving multithreaded adaptations, the program behavior is the result of the collaborative behavior of multiple threads and software components, and thus its formal analysis is required.

4.1 An Algebraic framework for the verification of context-aware adaptive systems

In this section we present an algebraic framework for modeling and verifying context-aware adaptive systems. This framework exploits the expressing power of the OTS/CafeOBJ method, the support for behavioral object composition and for parametric modules. In this way, it enables reasoning about the behavior of the components of such a system with respect to their context and about the system's adaptation in a consistent way. It can be used as a reference model and for verifying that a design of such a system satisfies safety properties.

The increased need of people to be connected everywhere and at all times, has resulted in a demand for intelligent applications that can sense and react to the changes in their requirements, users preferences and their environment. Such systems can be met in the literature under the term pervasive, self-adaptive or context-aware systems.

The term pervasive refers to the seamless integration of devices into the users' everyday life. Context-aware systems are able to adapt their operations to the current context without explicit user intervention and are strongly connected with self-adaptive systems. This fact has created two research areas; researchers in context-awareness focus on how to model and manage the context information, while researchers in self-adaptation are more concerned with how the system adapts and they usually separate the system's functionality from its management. This choice can increase the complexity of the system and calls for a more effective and reliable design.

In addition, nowadays there is a strong interest in the functional correctness of these systems as they are increasingly realized for safety-critical domains that include bank systems, healthcare and emergency scenarios. On the other hand, formal methods appear as a natural candidate to study the behavior of context-aware adaptive systems as a whole and to ensure their proper behavior and reliable development as they can discover design errors at an early stage of development.

However, these systems present new challenges compared to traditional systems and also addressing context awareness and self-adaptation in a consistent and integrated manner is not a trivial task. In particular, context-aware adaptive systems need to a) support the ability to monitor their environment, b) be flexible and c) dynamically adapt their behavior in response to changes in environmental conditions and in the behavior of the entities of the system. The latter changes can be induced by failures or unavailability of parts of the software system itself for example. In these circumstances, it is necessary for the system to re-adjust and continue achieving its purpose.

4.1.1 Proposed framework

To address the challenge of modeling context-aware adaptive systems in a consistent way, we want our framework to *capture the characteristics* of such systems naturally. Also, in order to satisfy the need for reliability in their development we want to be able to *express the systems' requirements*, and to *verify formally that their design satisfies the safety properties derived from the requirements*. We explain here how the design principles of our framework are in accordance with these goals. Some general characteristics that context-aware adaptive systems share are the following:

- One or more entities have a common environment/context
- The entities' behavior depends on the context
- The entities can interact with each other in order to perform a task
- The changes in the context can affect the behavior of the entities
- The actions of the entities could change the properties of the context

From these characteristics it is clear that the states of both the entities and the context change. The whole system adapts itself as well to cope with such changes. Thus, we argue that a context-aware system must be defined as a dynamic system (i.e. system with changing states). Also, we wish our framework to be able to observe the environment and to support flexibility. The latter can be enhanced if the refinement of one component does not affect the design and properties of the whole system. Thus we believe that both the entities and the context should be defined as separate systems. Additionally, we argue that the interaction of these components should be defined at a higher level so that it can be expressed in a clear way independently of a specific implementation of the context or the entities. For this reason, we believe that the whole system should be modeled as hierarchical composite system. Finally, the specification of such systems should be defined in a language that supports its formal verification. To achieve the above goals we propose a framework, whose architecture is shown in the following figure. In particular:

1. We introduce Context and Entity OTSs that specify the context and entities respectively.
2. We define a higher (meta-) level OTS with these objects as components (Context and Entity OTSs), using the methodology of behavioral object composition. This higher level OTS, called System, describes the full functionality of the context-aware adaptive system. The components' interactions and the system's adaptation are formally defined at this higher level.

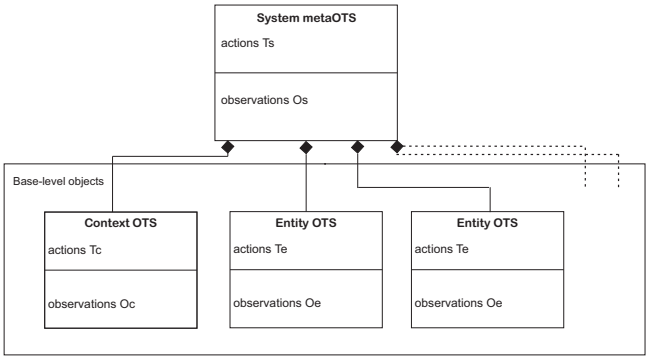


Figure 6: The proposed framework for an abstract context-aware adaptive system. The system is composed by a context and a possibly unbounded number of entities (represented by the dotted lines).

3. Finally, we specify these OTSs in CafeOBJ algebraic specification language that allows the formal verification of desired safety properties using theorem proving techniques.

Context OTS. We define the Context OTS which contains all the information that can be used to describe the environment of the entities. This information is modeled by observation operations, denoted by O_c that given a state of the context return a data type. Intuitively they can be thought of as experiments that can be used to characterize the state of the context, i.e. two states are considered equivalent if they return the same results for all possible experiments. During the development of the system this information could be mapped to any implementation, as ontologies for example. Finally, in order to describe how and under which circumstances the state of the context changes, we use a set of transition operations denoted by T_c.

Entity OTS. We define the Entity OTS which specifies an entity in a similar way, where with O_e we denote the set of observation operators that characterize the state of an entity and with T_e the transitions which correspond to the actions that change its state. The key distinction between a Context OTS and an Entity OTS is that the state of the entities depends on that of the context. However, sometimes it is difficult to identify when an observation should be part of the context and when it should be part of an entity. In most cases the following rule can be applied; the observations that cannot be affected by the entities but affect them, are part of the context.

System OTS. We define the System OTS which describes the meta-level where the reasoning for the behavior of the entities with respect to their

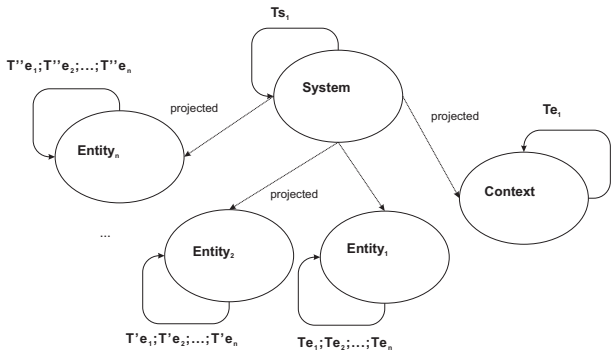


Figure 7: Graphical representation of the behavioral composition of OTSs

context is enabled. For this reason, it is defined as a composed OTS[7] with components a Context OTS and an arbitrary (and unbounded) number of Entity OTSs (figure 7). Thus, the state of the System fully depends on the states of its components. The System's observers O_s, depend on the values of O_c and O_e. The System's transitions T_s, correspond to the actions that change its state and define the system's adaptation. They also define how the changes on the state of the context affect the states of the entities and vice versa. This interaction is described through projection operations that "project" the composed System state to the states of its components and relate the transitions T_s with T_c and T_e. The communication between the components is defined in a clear way by using equations at the level of the specification of the composed object. *In this way, we can naturally capture the relationships between the components of a complex context-aware adaptive system.* In figure 8, an example of the interaction between the OTSs is presented graphically. The System transition τ_{s1} changes the state of all components; it is projected to the transition τ_{c1} of the Context OTS and to the sequence of transitions $\tau_{e1}; \tau_{e2} \ldots; \tau_{en}$, $\tau'_{e1}; \tau'_{e2} \ldots; \tau'_{en}$, $\tau''_{e\,1}; \tau''_{e\,2} \ldots; \tau''_{e\,n}$ of the Entity OTSs. In the table below the observers and transitions of the proposed framework are shown.

Table 6: Abstract OTSs of the framework.

Context OTS	Entity OTS	System OTS
$*[Csys]*$	$*[Esys]*$	$*[Sys]*$
$T_{ci} : Csys \quad D_1 \ldots D_n \to Csys$	$T_{ei} : Esys\, D_1 \ldots D_n \to Esys$	$T_{si} : Sys\, D_1 \ldots D_n \to Sys$
$O_{ci} : Csys\, D'_1 \ldots D'_n \to D'$	$O_{ei} : Esys\, D'_1 \ldots D'_n \to D'$	$O_{si} : Sys\, D'_1 \ldots D'_n \to D'$

Composition of OTSs. The way a composite object is defined mainly

[7]For details on how the composition of OTSs is still an OTS see [64], definition of composed OTS.

depends on the connection between its components. So to define the System as the behavioral composition of the Entity and Context OTSs we have to take into account that;

- The state of the entities depends on that of the context (and maybe vice versa), i.e. there exists *synchronization* between them.
- The *configuration* of context-aware adaptive systems may change, denoting that entities may malfunction and be removed or that simply entities enter/exit the context we are examining.

Thus, we use synchronized parallel composition which supports both synchronization and dynamic composition [15]. Dynamic composition allows an arbitrary number of component objects. This means that a dynamic object can be created and deleted in a composed object and its initialization is done with appropriate data playing the role of object identifier [16]. In particular, in order to retrieve the states of the components of a context-aware adaptive system, we define the following operators.

- Sys denotes the state space of the System object S,
- Id denotes the set of identifiers for the entity objects,
- $Esys$ denote the states of the entity objects, E_n and
- $Csys$ denote the states of the context object, C.

Projection operations are subject to some conditions (see [14] for details, definitions 1 and 4), which can be informally summarized as follows:

- All actions of the compound object are related via the projection operations to actions in each of the component.
- Each observation of the compound object is related via the projection operations to an observation of some component.

Finally, Context and Entity OTSs are specified as base level objects i.e. objects without projection operators.

Parametric System module. Taking a step further, we define the *core functionality* of a context-aware adaptive system in such a way that it can be *reused* for the modeling of different systems that share the same principles. In this way, part of the specification effort can be reduced. This is achieved mainly by exploiting CafeOBJ's support for parametric modules.

We have created a small library that can be expanded by the users to specify a specific context-aware adaptive system. In this way the core functionality of the system will be independent of the details of the particular

implementation and the components of the specified system can be refined without needing to change the whole specification. Also, a system specified by extending the library will satisfy a minimum set of context-aware characteristics.

More precisely, we have defined abstract theories that denote some elementary functions for entities and context objects (in the modules Entity and Context OTS respectively). These capture the following characteristics of a context-aware system; at the Context OTS we have defined appropriate operations and axioms for inserting, removing an entity from the context and for observing if an entity belongs to an instance of that context. At the Entity OTS, we have defined action and observation operators for describing a possible failure and a recovery of an entity and for denoting the passing of time (using a time advancing transition, a clock observer and a module representing the time).

The abstract theories are then used to compose a higher-level object (System OTS), which is *parameterized* by them. The projection operators describe the interaction between the Entity and Context OTSs ($cont : Sys \rightarrow Csys$ and $entity : Sys\ NzNat \rightarrow Esys$, where $Nznat$ is a predefined module denoting non-zero natural numbers and we used to identify the entity objects). Using them, the initialization of all objects is defined (*entity(init,N1) = initesys* and *cont(init) = initcont*) and the synchronization of the Entity clock with that of the System. Also, the System's actions are related to those of its components. For example the System's transition $Fail$ is related to the Entity's transition $fail$ (of the particular entity) and to the Context's transition *delEntity* (as it should be removed from the context). Similarly, the System's transition *Recover* is related to the Entity's transition *recover* and to the Context's transition *addEntity*. In CafeOBJ, the module System, with parameters the Entity and Context OTSs, is defined as: *System(X :: Entity, Y :: Context).*

Context and Entity OTSs can then be extended by the users to capture the behavior of a specific context-aware adaptive system (Extended Entity, Extended Context). The signatures of the modules Extended Entity and Extended Context (which import the predefined modules Entity and Context respectively and inherit their behavior) are shown in the following table; where, additional details of the particular implementation should be added - required data types, transitions and observers.

Next, the user-defined objects are used to instantiate the System OTS module (which can also be extended). This is defined in CafeOBJ terms as; $mod*\ ExtendedSystem\ \{pr(System(ExtendedEntity,\ ExtendedContext)\}$. In this way, the resulting theory preserves the composition between the extended components. This methodology is presented graphically in the figure below. We believe that even though the size of this library is small it can help users to model a context-aware system by providing the skeleton of the specification and more importantly the organization/hierarchy of the

Table 7: Extended OTSs of the framework.

ExtendedEntity OTS	*ExtendedContext OTS*
$mod*$ *ExtendedEntity*	$mod*$ *ExtendedContext*
$pr(Entity)$	$pr(Context)$
...	...
$f : \rightarrow Esys$	$d : Csys \rightarrow Bool$
– axioms for operator f	– axioms for operator d
...	...

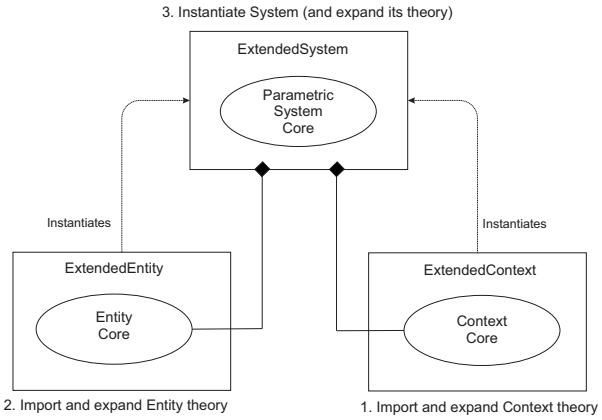


Figure 8: Steps to be taken on order to create the specification of a context-aware adaptive system.

components of their system. The library can be found at [48].

We have chosen to use the formalism of Observational Transitions Systems (OTS) for the specification of context-aware adaptive systems and the CafeOBJ algebraic specification language for the verification of their behavior due to the characteristics presented above (object composition, parametric modules) and the fact that it is based on equational logic, which is easier to learn than other logics such as higher order logic, because replacing equals by equals is part of everyday reasoning [65]. This method has been effectively applied to the analysis of different systems [65,66] and we believe that it is suitable for analyzing complex context-aware adaptive systems.

4.1.2 A traffic monitoring system

Intelligent transportation systems (ITS) are a worldwide initiative to exploit information and communication technology to improve traffic. One of the challenges in this area is the effective monitoring of traffic. In [67, 68] a monitoring system that provides information about traffic jams has been introduced that can be used by different kinds of clients such as traffic light controllers, driver assistance services, and so on. This system adapts

itself dynamically according to the changing traffic conditions and possible malfunctions of parts of the system and is a context-aware adaptive system. In order to demonstrate the effectiveness of the proposed framework we applied it to this traffic monitoring system.

The system consists of a set of intelligent cameras which are situated in a road. These cameras monitor and detect traffic jams and communicate with each other in order to report possible failures. Thus the system's behavior depends on both the behavior of the cameras and the road. Each camera has a limited viewing range and cameras are placed to get an optimal coverage of the road with a minimum overlap. An example of the highway is shown in figure 10, where cars move in one direction only. The main functionality of this traffic monitoring system is:

1. inform clients about traffic jams in a decentralized way, and
2. make the system robust to camera failures

To realize a decentralized solution, when traffic jam spans the range of multiple cameras they collaborate in organizations that provide information about traffic. These organizations have a master/slave structure and they split when the traffic resolves. The master of each organization is responsible for merging and splitting that organization by synchronizing with its slaves and for informing interested clients about the traffic jams. For these reasons, the master uses the *context information* provided by its slaves about their monitored traffic conditions. Each camera may have one of the above roles: master of a single member organization, master of an organization with slaves, or slave in an organization. Initially, all the cameras start as masters of single member organizations.

Depending on the role of each camera and the traffic conditions, the organizations may be adapted. If a master of a single member organization camera detects traffic it sends a "merge organization" request to its neighboring camera. If the neighbor does not detect traffic, organizations are not changed. If traffic is jammed and the neighbor is a master with slaves, the camera joins the organization and becomes master with slaves (figures 11 and 12). If the neighbor is a slave the camera joins the organization as a

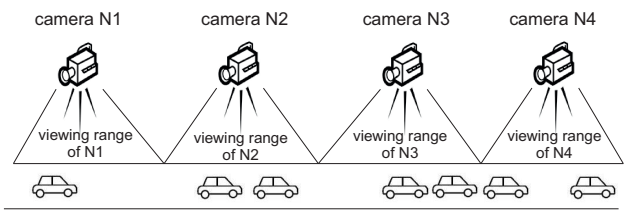


Figure 9: An example of the road with the cameras and their viewing ranges.

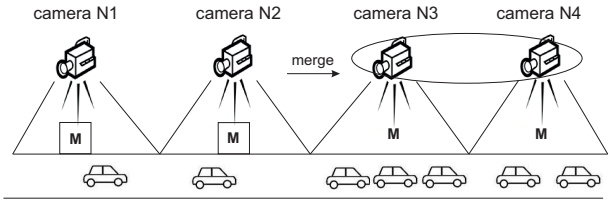


Figure 10: Camera N2 sends a merge request to camera N3 which is a master with slaves.

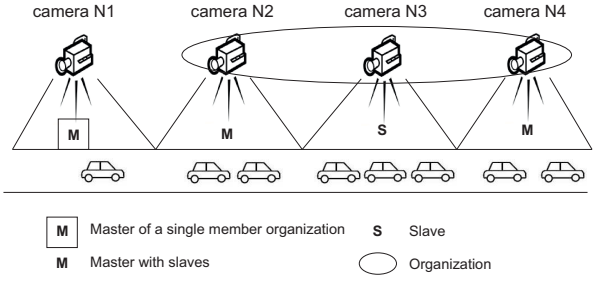


Figure 11: Camera N3 accepts the request and an organization with cameras N2, N3 and N4 is formed.

slave as well. Finally, if both are masters of a single member organization, the camera which sends the request becomes the master with slaves of the joined organization while the other becomes a slave.

In order to operate properly each camera is dependent on a particular set of cameras. A master of a single member organization is dependent on its neighboring nodes, a master with slaves is dependent on its slaves and its neighbors and a slave is dependent on its master and its neighbors. To ensure that the system will be fault tolerant, a *coordination protocol* that uses ping-echo messages is used. Each camera sends ping messages to its dependent cameras. Wait time indicates when a camera should respond with an echo message after a ping message has been sent to that camera. A camera reaches timeout when the time exceeds the last ping time for that camera plus the wait time, and in that case adaptation of the system is required.

Formal specification. To specify this context-aware adaptive system in our framework we first identify that the cameras correspond to entities which are situated in a context. This context contains information about the road and the traffic. We assume that we have an arbitrary number of cameras and that the road is divided into viewing ranges so that each camera has one such range (figure 10). After importing the library modules

Context, Entity and System OTSs we create the Extended Context, Entity and System OTSs, by adding details of this traffic monitoring system.

The *ExtendedContext OTS* specifies the traffic on the road. It needs to specify the number of cars on a viewing range at any given state and how these cars enter, exit the road and move along the viewing ranges. The observers and transitions used to define it can be found in table 8. We assume that initially there are zero cars in the road. This is declared in CafeOBJ terms as;

```
numberofcars(initcont,N2) = 0.
```

In this equation, N2 represents an arbitrary range of the road and the operator *initcont* represents the initial state. The effective condition of the *movecar* transition is declared in CafeOBJ as;

```
c-movecar(C,N2,N3) = (s(N2) = N3)  and not(numberofcars(C,N2)
= 0).
```

This equation states that a car moves from range $N2$ to $N3$ if these ranges are subsequent and the number of cars at $N2$ is not zero. The result of the transition *movecar* is defined as follows;

```
numberofcars(movecar(C,N2,N3),N1) = (numberofcars(C,N1)+1)
if c-movecar(C,N2,N3) and (N1 = N3).

numberofcars(movecar(C,N2,N3),N1) = sd(numberofcars(C,N1),1)
if c-movecar(C,N2,N3) and (N1 = N2).
```

These equations denote that the number of cars at range $N3$ increases by one and at range $N2$ decreases by one, after the successful application of the transition.

In table 8 $NzNat$ is used to identify the viewing ranges. *initcont* and each transition are constructors of Sys, which corresponds to R_S. R_S is the type denoting the set of all reachable states wrt S. Also Sys denotes R_S but not Y if the constructor-based logic is adopted, which is the current logic underlying the OTS/CafeOBJ method.

In table 8 *Bool* is a (built-in) visible sort representing the truth values and $Setofnat$ denotes a set of natural numbers.

Table 8: Observers and transitions of the ExtendedContext OTS.

Observers	
$numberofcars : Csys\ NzNat \rightarrow Nat$	Returns the number of cars at a specific viewing range
Transitions	
$releasetraffic : Csys\ NzNat\ NzNat \rightarrow MCsys$	Models the feeding of a viewing range with cars
$movecar : Csys\ NzNat\ NzNat \rightarrow Csys$	Models the movement of a car along subsequent viewing ranges

Table 9: Observers of the ExtendedEntity OTS.

Observers	
$sendpingtimer : Esys\ NzNat \rightarrow Time$	Stores the time where a ping message is sent to a camera
$timeout : Esys\ NzNat \rightarrow Bool$	Becomes true when the timeout of another camera is observed
$mymaster : Esys \rightarrow NzNat$	Returns the master with slaves
$myslaves : Esys \rightarrow Setofnat$	Returns the set of slaves
$received : Esys\ NzNat \rightarrow Bool$	Becomes true when a ping message is received
$sent : Esys\ NzNat \rightarrow Bool$	Becomes true when an echo message is received
$cansend : Esys\ NzNat \rightarrow Bool$	Becomes true when a ping message can be sent to another camera

The *ExtendedEntity OTS* must express the full functionality of a camera, i.e. denote the role of each camera in an organization, and model the sending and receiving of messages and so on. For the definition of its state space the observers of table 9 were used.

The observer *mymaster*, which returns the current master with slaves of the entity, returns the constant `null` if the camera is a master of a single member organization (i.e. has no master with slaves). If the camera is a slave, *mymaster* returns the identification number of its master with slaves and finally if it is a master with slaves it returns its own id. To specify how the state of a camera changes the transitions of table 10 were used. The definition of the timeout of a camera is given below in CafeOBJ notation;

```
timeout(tick(E,T1),N2) = true if (now(E) + T1 >
sendpingtimer(E,N2) + waitime and sent(E,N2) = false.
```

The above equation states that a timeout of camera $N2$ is observed after the application of transition *tick* if the time exceeds the time where the last ping message was sent to $N2$ plus the *waitime* (which is a CafeOBJ constant of sort time) and no echo message from $N2$ has been received.

The *ExtendedSystem OTS* defines how the cameras interact and behave based on their context, to provide an effective monitoring of traffic. The observers that were used to define its state can be seen in table 11.

In this table the visible sort *status* represents the traffic and $free$ and *congested* are two constants of this sort. *Client* is a visible sort representing an arbitrary client interested in the traffic conditions of the road. *Content* models the content of the message that is sent to a client when he/she asks for information.

The transitions that change the state of the ExtendedSystem can be seen in table 12. In the following, we will give some details for the transitions

Table 10: Transitions of the ExtendedEntity OTS.

Transitions	
$changemaster : Esys\ NzNat \rightarrow Esys$	Changes the current master with slaves of the entity
$addslaves : Esys\ Setofnat \rightarrow Esys$	Represents the addition of a set of slaves to the current set of slaves
$removeslave : Esys\ NzNat \rightarrow Esys$	Represents the removal of a slave from the current set of slaves
$removeallslaves : Esys \rightarrow Esys$	Removes all the slaves of the entity directly
$sendping : Esys\ NzNat\ Time \rightarrow Esys$	Models the sending of a ping message to another camera
$receiveping : Esys\ NzNat\ Time \rightarrow Esys$	Models the receiving of a ping message from another camera
$sendecho : Esys\ NzNat\ Time \rightarrow Esys$	Models the sending of an echo message to another camera
$receiveecho : Esys\ NzNat\ Time \rightarrow Esys$	Models the receiving of an echo message from another camera

Table 11: Observers of the ExtendedSystem OTS.

Observers	
$trafficstatus : Sys\ NzNat \rightarrow status$	Returns the traffic status of a camera
$request : Sys\ Client\ NzNat \rightarrow Bool$	Becomes true when a client asks a camera for information
$message : Sys \rightarrow Content$	Returns the content of the message that is sent to the client
$leftneighbor : Sys\ NzNat \rightarrow NzNat$	Given a camera, returns its left neighboring camera
$rightneighbor : Sys\ NzNat \rightarrow NzNat$	Given a camera, returns its right neighboring camera

textitpingmessage, *detectfailure*, *monitortraffic* and *orgrequest*, which define the main functionality of the system.

The transition *pingmessage* is used to describe the ping-echo coordination protocol (the first $NzNat$ argument in the signature denotes the sender and the second the receiver). When the transition $pingmessage(M, N1, N2)$ is applied successfully in an arbitrary system state S, this corresponds to camera $N1$ sending a ping message to camera $N2$. Thus, it is related with the application of the transition *sendping* to camera $N1$ and *receiveping* to camera $N2$. This is defined using the appropriate projection operator with the following equations;

```
entity(pingmessage(S,N1,N2,T1),N3) = sendping(entity(S,N1),N2,T1)
if (N3 = N1).
```

Table 12: Transitions of the ExtendedSystem OTS.

Transitions	
$pingmessage : Sys\ NzNat\ NzNat\ Time \rightarrow Sys$	Represents the exchange of ping messages
$echomessage : Sys\ NzNat\ NzNat\ Time \rightarrow Sys$	Represents the exchange of echo messages
$detectfailure : Sys\ NzNat\ NzNat \rightarrow Sys$	A camera detects the failure of another camera
$monitortraffic : Sys\ NzNat \rightarrow Sys$	A camera monitors the traffic jam
$orgrequest : Sys\ NzNat\ NzNat \rightarrow Sys$	Represents the merge of the organizations
$askinfo : Sys\ Client\ NzNat \rightarrow Sys$	A client asks a camera information about its traffic conditions
$sendinfo : Sys\ Client\ NzNat \rightarrow Sys$	The camera sends the requested information to the client

```
entity(pingmessage(S,N1,N2,T1),N3) = receiveping(entity(S,N2),
N1,T1) if (N3 = N2).
```

The transition $detectfailure$ defines the adaptation of the system when a camera fails, as the system should be able to operate correctly after a failure although the traffic state in the range of the failed camera is no longer monitored. This transition may have four different outcomes according to the role of each camera. In the first case, if a master of a single member organization detects the failure of another master its role stays the same after the application of the transition. In the second case, if a master with slaves detects the failure of one of its slaves its role does not change. But, if the slave that fails is the only slave of the organization, its master with slaves will become master of a single member organization. Finally, if a slave detects the failure of its master, it will become master of a single member organization after the application of the transition. The effective condition of $detectfailure$ can be seen below;

```
c-detect(S,N1,N2) = timeout(entity(S,N1),N2) and not(failure
(entity(S,N1))).
```

It states that the camera that detects the failure must not have failed and the camera that fails must have reached a timeout. The result of the successful application of $detectfailure(S, N1, N2)$ in an arbitrary system state S is projected to the state of the arbitrary camera N3 by the following equations:

```
entity(detectfailure(S,N1,N2),N3) = changemymaster
(entity(S,N3), null) if (N3=N1) and c-detect(S,N1,N2) and
(mymaster(entity(S,N1))=N2) and (mymaster(entity(S,N2))=N2) .
```

```
entity(detectfailure(S,N1,N2),N3) =  entity(S,N3) if
(N3=N2) and c-detect(S,N1,N2) .

entity(detectfailure(S,N1,N2),N3) =  entity(S,N3) if
not (N3=N2) and not (N3=N1) and c-detect(S,N1,N2) .
```

The first equation states that the transition *changemymaster* is applied to camera $N3$, if it is equal to $N1$, and changes its master with slaves to null (i.e. becomes master of a single member organization) if the effective condition of the transition holds and if in the previous state $N1$ was a slave of $N2$ and $N2$ was master with slaves. The state of camera $N3$, if it is equal to $N2$, does not change as declared with the second equation. Finally, if camera $N3$ is neither $N1$ nor $N2$ its state stays the same (third equation).

The monitoring of the traffic by the cameras is denoted by the transition *monitortraffic*. This transition is a good example of how the state of the ExtendedSystem OTS depends on the state of the ExtendedContext OTS. When *monitortraffic* is applied this interaction is defined using the projection operator `cont`, as shown below;

```
trafficstatus(monitortraffic(S,N1),N3) = free if
(numberofcars(cont(S),N1) <= n) and
c-monitor(S,N1) and (N3 = N1) .

trafficstatus(monitortraffic(S,N1),N3) = congested if
(numberofcars(cont(S),N1) > n) and
c-monitor(S,N1) and (N3 = N1).
```

In the above equations, the observer *trafficstatus* of the ExtendedSystem OTS returns the constant `free` as the traffic status of a viewing range, if the observer *numberofcars* of the ExtendedContext OTS on that range returns a value below a constant n (and congested otherwise).

Finally, the transition *orgrequest* defines the adaptation of an organization when a new camera joins it. Assume that camera $N1$ sends a "merge request" to camera $N2$. If $N2$ also detects traffic then they will be merged in one organization. In CafeOBJ terms this is defined as follows;

```
entity(orgrequest(S,N1,N2),N3) = changemymaster(entity(S,N3), N1)
if c-org(S,N1,N2) and mymaster(entity(S,N2)) = null and (N3 = N2)

entity(orgrequest(S,N1,N2),N3) = addslaves(changemymaster
(entity(S,N3),N1),N2) if (N3 = N1) and c-org(S,N1,N2) and
mymaster(entity(S,N2)) = null
```

These equations denote how the states that correspond to cameras $N1$ and $N2$ change. Camera $N2$ becomes a slave with $N1$ as its master with slaves (first equation) and camera $N1$ becomes master with slaves and also adds $N2$ to its list of slaves (second equation) if the effective condition of the transition holds and if in the previous state both cameras were master of single member organizations. But in the effective condition of *c-org(S,N1,N2)* there exist the sub-conditions $trafficstatus(S, N1)$

$= congested$ and $trafficstatus(S, N2) = free$ and $trafficstatus$ depends on the observer $numberofcars$ of the ExtendedContext OTS. Thus, the states of the entities depend on the state of the context, i.e. this transition exhibits entity-context dependency. We believe that it is important for a framework for context-aware adaptive systems to support both Entity-Context and System-Context, dependencies. Our framework can naturally express them as was exhibited by the definition of transitions $orgrequest$ and $monitortraffic$, respectively.

Formal verification. A framework that is used for the specification of context-aware adaptive systems must be equipped with verification techniques for ensuring their correct behavior. This need becomes even bigger when we are designing critical systems where it can be very costly to correct possible errors and it is important to detect them before the implementation of the system's specification.

We demonstrate here how our framework can be used to verify the specification of context-aware adaptive systems, by applying the OTS/CafeOBJ method to the traffic monitoring system presented in the previous section.

In table 13 the properties that were verified for the traffic monitoring system are presented. Invariants 1 and 2 concern the proper communication between two cameras via the ping-echo protocol. In particular, the first property expresses the fact that the system should not be in a deadlock. A deadlock for example may occur when two cameras have sent ping messages to each other and wait for responses, without being able to send any messages (figure 13). Thus, we state that two cameras cannot have received ping messages from each other at the same time. The second property declares that if a camera has received a ping message from another camera then it cannot send a ping message to that camera. Finally, the third property states that it is not possible for a camera to be a slave of another camera and the second camera not to have the role of master with slaves, meaning that a slave and its master with slaves appear always together. This is an important property for the system because if we had only slaves without masters the interested clients could not be informed about the traffic conditions.

To prove such properties with the OTS/CafeOBJ method, four steps need to be taken, as we have already mentioned. To explain better the verification process, we will use invariant 1 as our running example.

1. The first step is to express the property as a predicate in CafeOBJ terms in a module, usually called INV.

```
inv1 : Sys NzNat NzNat -> Bool
inv1(S,N1,N2) = not(received(entity(S,N1),N2) and
received(entity(S,N2),N1)) .
```

2. The next is to define the inductive step in a module (usually called ISTEP), i.e. a predicate which states that if the property holds in an

Table 13: Invariant properties of the traffic monitoring adaptive system.

	Definition in CafeOBJ terms
1.	$not(received(entity(S, N1), N2)$ & $received(entity(S, N2), N1))$
2.	$not(received(entity(S, N1), N2)$ & $cansend(entity(S, N1), N2))$
3.	$not(N1\ /in\ myslaves(entity(S, N2)$ & $not(mymaster(entity(S, N2)) = N2))$
	Informal description of the properties in natural language
1.	It is not possible for two cameras to have received ping messages from each other at the same time.
2.	It is not possible for a camera to send ping message to another camera if it has received a ping message from that camera.
3.	It is not possible one camera to be slave of another and the second camera not to be master with slaves.

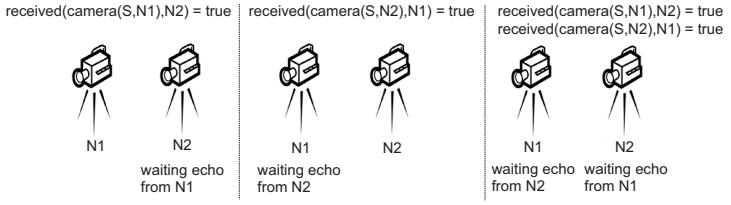


Figure 12: A case where the cameras are in a deadlock.

arbitrary state, say s, then that implies that it holds in any successor state, say s'.

```
istep1 : -> Bool
istep1 = inv1(s,n1,n2) implies inv1(s',n1,n2) .
```

3. The third step is to ask CafeOBJ to prove via term rewriting (using the reduce command), if the property holds for an arbitrary initial state.

```
open INV
red inv1(init,n1,n2) .
close
```

4. Finally, s' must be instantiated and then ask CafeOBJ to prove the inductive step for each transition rule. For example for the transition *pingmessage* this is declared as follows;

```
open ISTEP
s' = pingmessage(s,n3,n4,t1) .
red istep1 .
close
```

When we ask CafeOBJ to prove a property three results might be returned by the system: *true*, $false$ and a CafeOBJ *term*. If true is returned

Table 14: Case splitting in proving invariant 1.

Case 1	Case 2
$received(camera(s, n3), n4) = false$	$received(camera(s, n3), n4) = true$
CafeOBJ returns false	CafeOBJ returns *true*.
red $inv2(s, n3, n4)$ *implies* $istep1$.	Proved.
CafeOBJ returns *true*.	
Proved.	

the proof is successful. For example, CafeOBJ returned true for the initial state. If a CafeOBJ term is returned, different to true or false, then there exist some terms that the system cannot fully reduce. The user must intervene and split the case, by stating that the returned term equals to true and false in turn (computer-human interactive method). This creates two new proof obligations and is known as case splitting. The case defined by the following proof passage, returned the term $received(entity(m, n3), n4)$, for the transition *pingmessage*.

```
open ISTEP
-- c-ping(s,n3,n4) = true.
failure(entity(s,n3)) = false.
failure(entity(s,n4)) = false.
cansend(entity(s,n3),n4) = true.
mymaster(entity(s,n3)) = n3.
mymaster(entity(s,n4)) = n3.
n1 = n3.
n2 = n4.
s' = pingmessage(s,n3,n4,t1).
red istep1 .
close
```

Thus, to help CafeOBJ reduce the returned term, we had to split it in two subcases. The first, where $received(camera(s, n3), n4) = false$ returned true. But the second, where $received(camera(s, n3), n4) = true$ returned false.

In this case, where CafeOBJ returns false, two things might hold. The property does not hold for our system, or the case that returned false is unreachable. In the first case we have found a counter example and it might be necessary to redesign our system. In the second case we must create a lemma, a new invariant, which states that the case that returned false is not reachable. Of course the lemma must be proved separately. As the case returned in our example was unreachable for our system, we used invariant 2 to discard it (Table 13). Following this methodology, we successfully verified that the specification satisfies the invariant properties of table 13. The full specification of the traffic monitoring system and the proofs of the invariants can be found at [48].

Some existing methodologies for specifying context-aware and self-adaptive systems, include the work of Weyns et al. in which a formal reference model,

called FORMS, is presented for describing and reasoning about the key characteristics of system's adaptation [68]. Coronato et al. propose an extension of ambient logic and ambient calculus for specifying the requirements of critical pervasive applications. Also they discuss some issues concerning runtime testing and static verification using model-checking [69]. In another approach, Khakpour et al. use dynamic policies to govern and adapt system's behavior [70].

Few approaches targeting the verification of these systems exist. Schneider et al. propose a technique to extract an abstract model of the adaptation behavior of such systems, specify the desired properties using temporal logics and model-check them. To represent infinite state systems, finite automata are used, however termination cannot be guaranteed as the authors state, due to the undecidability of most verification problems [71]. Hussein et al. present a method for specifying and verifying context-aware software systems, in which the model is transformed to Petri Net and then the verification is performed with the Romeo tool. Also, they classify the possible system variations into dependent and independent for reducing the possible system states and the transition between them, making the state explosion problem not easily reached [72]. Adler et al. propose a framework for verifying the system's adaptation based on temporal logics and the Isabelle/HOL prover that allows the combination of theorem proving and model checking techniques. However, they focus on model-checking as they believe that using a theorem prover can be tedious [73]. Iftikhar et al. report a first step on modeling adaptive decentralized systems using timed automata. A case study is presented where the desired properties are specified in timed computation tree logic and verified using the Upaal tool [67]. Finally, Rakib et al. give ontological representation of contexts, use Horn-clause rules to build their rule-based system and verify properties using Maude's model-checker. Their approach is focused in modeling the communication between agents and in the verification of time resource-bounded properties [74]. Our framework addresses both the specification and verification of context-aware and self-adaptive systems and is based on Observational Transition Systems written in CafeOBJ, a different formalism than those adopted in the above approaches.

Our framework compared to existing approaches has the following contributions. First, while other approaches that use model checking techniques [67, 69, 71, 72, 74] cannot model an unbounded number of context aware agents or may suffer from the state explosion problem, our approach has the ability to deal with systems that have an infinite number of states and a variable number of agents.

Second, model checking often does not support understanding of the specified system since it only gives "yes" or "no with counter example", in contrast with our approach which returns enough feedback to the user and thus helps him/her understand better the system's design.

Third, the OTS/CafeOBJ method compared to methodologies that are based on theorem provers such as Isabelle/HOL [73] has been argued to have the following advantages: (1) balanced human-computer interaction and (2) flexible but clear structure of proof scores. The former means that humans focus on proof plans, while computers deal with tedious and detailed computations. For example humans do not necessarily have to know which equations or rules should be applied to the proof. The latter means that lemmas used in the proof do not need to be proved in advance and that results obtained by case analyzes and lemmas discovered are explicitly and clearly written in proof scores [53].

We also discuss here how our framework supports the need of such systems to realize context-awareness and flexibility, in more details. *Context-awareness*, i.e. the ability of the system to monitor the environment, is expressed in our framework via the compositionality of the OTSs. The environment of the entities is modeled as a separate system, the Context OTS, whose states can be observed by all other OTSs through projection operators. In [75] it is stated that *flexibility* is achieved by separation of concerns, computational reflection and component-based design. Separation of concerns is realized by separating the specification of the System composed OTS from the specifications of its components, i.e. the Context and the Entity OTSs. In this way, we are allowed to change the specification of the former without changing the specification of the latter. Computational reflection refers to the ability of a system to reason about, and possibly alter its own behavior. This feature is realized in each OTS system via its observation operators. These monitor its state and depending on their observable values the OTS can fire transition rules thus, changing its behavior. Finally, component based design is about composing loosely coupled independent components into systems and the ability of each component to adapt its behavior autonomously. This is achieved with the modular design of the system where the transitions of the base level objects can change their state independently of the higher level system.

However, the formal framework proposed here is not without limitations. The OTS/CafeOBJ approach to systems' analysis is less rigorous than analysis using other theorem provers like Isabelle/HOL [76] and Coq [77]. Proofs constructed under their support are basically guaranteed to be correct, while humans might overlook some cases to consider or proofs of some lemmas in the proof score approach because there are no tools available to check such human errors [53].

To solve this issue, some tools have been developed, in particular Gateau [78] and Crème [79]. Gateau generates proof scores given state predicates for case splitting and necessary lemmas. Crème is an automatic invariant verification tool for specifications of OTSs. To take advantage of model checkers, which are complementary to interactive theorem provers, two other tools have been developed, Chocolat/SMV [80] and Cafe2Maude [81]. Choco-

lat/SMV translates CafeOBJ specifications of OTSs into SMV specifications, which can be model-checked with SMV. Cafe2Maude translates CafeOBJ specifications of OTSs into Maude specifications that can be model-checked with the Maude's model checker. We are also working towards these directions by developing a tool that will automate the OTS/CafeOBJ verification method using the Athena proof system [66, 82] (see next chapter).

5 On Integrating Algebraic Specifications with Polymorphic Multi-Sorted First-Order Logic via Athena

Methods of software verification can be divided into formal methods, which rely on rigorous analysis of mathematical models of the system being checked and the desired properties; static analysis methods, which seek errors without running the software; dynamic analysis methods, which verify actual behavior of the system under study in some scenarios of its operation; and review or inspection, which is performed by experts based on their experience and knowledge.

All these methods have their advantages and disadvantages, different application domains, and their efficiency may differ significantly in different contexts. Valuable verification of large scale complex systems is better with the combined use of these methods, because their combination can overcome disadvantages of the individual methods.

In algebraic specification methods for example, a branch of formal methods area, systems are specified based on algebraic modeling, and then the specifications are verified against requirements using algebraic techniques. Algebraic specification languages such as CafeOBJ [12], Maude [83] and CASL [84] have well-known advantages for modeling and reasoning about digital systems. The specifications are relatively simple, readable and writable, and can be executed and automatically analyzed in various other ways to provide valuable information to the modelers.

However, specification languages have better results, when they are integrated with more conventional theorem proving systems. CASL, for instance, has been interfaced with HOL/Isabelle [85, 86], through HOL-CASL [87]. Also the Hets tool (Heterogeneous Tool Set [88, 89]) has connections with the algebraic specification languages Maude and CASL and external theorem provers (SPASS, Vampire, Darwin, KRHyper and MathServe).

In this chapter we propose a framework of integration of the CafeOBJ algebraic specification language with the Athena [82] interactive theorem prover, and a common interface that transforms OTS-based specifications written in CafeOBJ into Athena specifications. The aim of the integration of these two environments (not their semantics) is to exploit both the nice

properties of CafeOBJ specifications and the soundness and automation in verification offered by Athena. The proposed methodology integrates the two environments (not their semantics)

Algebraic specification methods are mainly concerned with providing support for the systematic development of correct programs from specifications with verified properties. In the OTS/CafeOBJ approach, CafeOBJ provides mechanized implementations of Observational Transition Systems (OTSs), a species of behavioral specifications, that allow users to specify distributed systems using multi-sorted conditional equational logic with sub sorting. The specifications are executable (via rewriting), which is useful for building up computational intuitions about the underlying system. In addition, CafeOBJ allows users to compose proof scores that establish certain invariant properties, typically by induction.

Athena on the other hand, is a system based on general polymorphic multi-sorted first-order logic. It integrates computation and deduction, allows for readable and highly structured proofs, guarantees the soundness of results that have been proved, and also has built-in mechanisms for general model-checking and theorem-proving, as well as seamless connections to state-of-the-art external systems for both.

By integrating these two methodologies we wish to combine the strengths of CafeOBJ, most notably succinct, composable, executable specifications based on conditional equational logic with those of Athena, namely, structured and readable proofs, soundness of the results and greater automation both for proof and for counterexample discovery.

5.1 Athena -First-Order Logic- proof system

Athena [82, 90] is an interactive theorem proving environment, with separate languages for computation and deduction. As a programming language, Athena is a higher-order functional language. As a theorem proving system, Athena is based on polymorphic multi-sorted first-order logic. For conventional programming, Athena has built-in domains like strings, booleans and numbers, but also lists, terms and sentences of (first-order, multi-sorted) logic, substitutions, and so on [90].

The main mechanism for program composition is the procedure call. Procedures are higher-order as they can take procedures as arguments and return procedures as output as well. The main tool for constructing proofs is the method call. A method call represents an inference step and can be primitive or complex. Methods can accept as arguments other methods and/or procedures, and thus are also higher-order.

The evaluation of a procedure call, if it does not raise any error, can result in a value of any type, while the evaluation of a method call can result only in a *theorem*: a sentence of logic that is derived by inference from axioms and other theorems.

Athena's methods are expressions of proofs. A key attribute of the Athena proof language is that such expressions of proofs are both machine-checkable and human-readable. The readability of Athena proofs is due to the natural way with which one can express important proof methods. In part, this is due to a fundamental mechanism of Athena, its assumption base.

The assumption base is a global set of sentences maintained by Athena. More precisely, the assumption base for a certain point in the proof contains all the facts that are known to be true at that point. Initially the system starts with a small assumption base, containing defining axioms for some built-in function symbols. When an axiom is postulated or a theorem is proved, the corresponding sentence is inserted into the assumption base. During a proof construction, Athena's methods can interact with the assumption base, checking to see if some arguments are present in the set and/or making new entries [90]. This seems to be a natural way of reasoning that differentiates Athena from other interactive proof systems.

Another distinctive characteristic of Athena is its connection with various external systems. Through its interfaces with powerful automated theorem provers like Vampire and SPASS, Athena provides significant proof automation[8]. Finally Athena's integration with those provers is seamless, meaning that the user does not have to leave Athena's high-level notation in order to call the external systems.

5.1.1 Athena's tools for OTSs verification

We present here several features of the methodology that can be used to better understand the specified system, model-check desired properties and verify them via theorem proving, both automatically and in a structured and more detailed way.

Simulation. In Athena rewriting is efficiently performed by compiling equational (and conditional equational) axioms into functional code. Programmatic access to that code is given via the unary procedure *eval*, which will take an arbitrary term t and will attempt to reduce it to a normal form t' (a term that contains only constructors) by executing the various compiled procedures associated with the axioms defining the function symbols that occur in t.

Based on *eval*, a procedure, *simulate*, takes a sequence of states $s_1, \ldots, s_n$, where s_1 is usually the initial state and each s_i for $i > 1$ is obtained by applying a transition operator to s_{i-1} and (if necessary) to automatically generated constants of appropriate sorts (depending on the signature of the

[8]Vampire [91] is a theorem prover for first-order classical logic developed in the University of Manchester and SPASS [92] is an automated theorem prover for first-order logic with equality developed in Max-Planck-Institut.

transition operator), and prints out each state in the sequence by applying all observer functions to the given state (plus other constants, if necessary, depending on the signatures of the observers), and specifically by evaluating the applications of those observer functions (by using the relevant axioms as rewrite rules).

Counterexample discovery. Counterexamples to conjectures (i.e., to conjectured invariants) in Athena can often be discovered automatically with the falsify procedure. Athena is also integrated with external systems that can be used for counter model generation, most notably SMT and SAT solvers, but falsify is a built-in facility and has the simplest interface: To falsify a conjecture p, we can use (falsify p N). Here N is the desired *quantifier bound*, namely, the number of values of the corresponding sort that we wish to examine (in connection with the truth value of p) at each quantifier of p.

When falsification fails within the given bound, the term *'failure* is returned. When falsification succeeds, it returns specific values for the quantified variables that make the conjecture false.

Automated Theorem Proving. A method for completely automated inductive reasoning is automatically defined whenever a new structure or datatype is introduced. The name of the method is the name of the corresponding structure joined with the string `-induction`, in lower case.

ATPs are not only available for proving inductive properties, but for proving any goal whatsoever, from any premises whatsoever. This type of more fine-grained application of automated theorem proving is provided by the language constructs

$$p \quad \texttt{from} \quad p_1, \ldots, p_n$$

and

$$p \quad \texttt{from} \quad L$$

where L is an arbitrary list of sentences. In both cases, the sentences on the right-hand side of `from` are the premises from which the derivation is to proceed, while the single sentence p on the left-hand side of `from` is the desired goal to be derived from the given premises. For even more fine-grained control one can use the primitive method `prove-from`, which allows for setting time limits and other parameters, and choosing which ATP to use (Spass is used by default).

Structured Proofs. Completely automated correctness proofs are usually difficult or impossible to obtain. Even when possible, a completely automatic proof will not shed much light on a system's workings. A structured

proof that derives the desired result in a piecemeal fashion can be much more valuable in *explaining* the underlying system, i.e., in explaining why a given property holds. In addition, the effort invested in constructing the proof often pays off in increased understanding, and also in the discovery of errors, unintended consequences of design constraints, and so on. Athena uses a Fitch-style formulation of natural deduction [93] which helps to make proofs and proof algorithms more perspicuous.

Proofs' Checking. Proofs in Athena have a formal evaluation semantics which guarantee that a proof which has been successfully evaluated (checked) is in fact *sound*. This can increase our confidence in the correctness of the result.

5.2 Proposed framework: Cafe2Athena, from CafeOBJ to Athena Specifications

The OTS/CafeOBJ approach presents many advantages as discussed in the previous chapters, the most important of which in our opinion and experience is that the proof score methodology can effectively guide the user to discover the required case splittings and occasional lemmas for the proof. However, we believe that this methodology could benefit when used in combination with the Athena proof system for the following reasons:

At first, each proof conducted in Athena is checked for soundness by the system. Thus, Athena could act as a validator for the proofs conducted in CafeOBJ. Also, Athena as we have already mentioned is integrated with some of the strongest automated theorem provers (ATPs) of the state of the art, like Vampire [91] and SPASS [92]. Thus, via Athena, access to these highly efficient ATPs can be enabled for OTS/CafeOBJ specifications as well, which could help with the automation of the verification process. Finally, Athena uses, a Fitch style, natural deduction proof system. Such types of proof structures are easy to follow and thus it could help with the understanding of OTS/CafeOBJ proofs by a wider audience.

For the above reasons we propose a verification methodology which combines both environments (CafeOBJ and Athena) which can be summarized to the following steps:

- Step 1: create the OTS specification in CafeOBJ
- Step 2: automatically, generate an equivalent Athena specification
- Step 3: attempt to falsify using Athena the property under verification. If unsuccessful proceed to the next step else either the specification or the property should be revised.

- Step 4: apply the proof score methodology in CafeOBJ until a lemma is required. Discover a candidate lemma in CafeOBJ that can discard the problematic case.

- Step 5: attempt to falsify the candidate lemma in Athena. If unsuccessful continue the proof score using the candidate lemma else either the specification or the property should be revised.

- Step 6: iterate steps 4 to 5 until the proof scores methodology is completed for the property in question and also for all the lemmas used.

- Step 7: using the insights gained by the proof scores (case splittings and lemmas) generate an Athena proof and check its soundness.

With the proposed methodology (steps 1 to 3), the user could save considerable time by first attempting to falsify the property in question. If indeed a counter example is returned by Athena then either the property in question is not invariant for the specification, or there might be a bug in the specification itself. In the first case, the proof is completed and the property falsified. In the second case the output of Athena usually provides sufficient information for the discovery and correction of the bug.

During the verification of complex systems with the proof score methodology, it is possible that the user will consider as lemmas, properties which while successfully discard the problematic cases, are not invariant for the specification. This usually is discovered at a late stage of the verification of the lemmas, in which case the proof needs to be recreated using a different candidate lemma. This can at times become an important time-sink for the verification. Athena, could potentially inform the user of the error in the candidate lemmas at a much earlier stage (steps 4 to 5) and thus save the user valuable time.

Also once the proof score methodology is completed, by transferring the insights gained by it to Athena we can easily create a formal proof for the property in question. Athena can thus inform us about the soundness of the proof or point to the errors of our reasoning.

Finally, as we mentioned earlier the final proof constructed in Athena will be in a style easy to understand and thus could help provide explanation as to why the property is invariant or not for the system, in other words act as a sort of documentation.

5.2.1 Rules of translation

In order to obtain an Athena specification from a specification written in the OTS/CafeOBJ method we have defined an appropriate translation schema. The basic units of OTS/CafeOBJ specifications and their transformation into Athena notation are shown in Table 15.

Table 15: The basic units of OTS/CafeOBJ specifications and their transformation into Athena notation.

OTS/CafeOBJ notation	Athena notation
tight modules	datatypes
loose modules	domains
state	structure
initial state	state constructor
state transitions	state constructors
observers	functions
equations	axioms

More precisely, in Athena initial algebras (with tight semantics) are specified using the keyword *datatype*, whereas an arbitrary carrier (with loose semantics) is introduced with the *domain* keyword. An induction principle is automatically generated for every new datatype. States are formalized as a *structure* and state transitions as its constructors. Structures are very much like datatypes, except that there may be confusion, i.e. different constructor terms might denote one and the same object. An induction principle is also automatically generated (and, of course, valid). We continue with the declaration of the observers. They are defined as functions with the constraint that they take as input a state (and maybe additional input) and return some datatype. The equations that define the initial state, as well as the pre- and post-conditions of the state transitions are defined as axioms. One last remark is that Athena variables, that will needed for the definition of the axioms, start with a question mark, but identifiers can refer to variables, i.e., variables are denotable values in Athena's programming language).

In more details, an arbitrary OTS specification written in CafeOBJ terms is translated into an Athena specification through the operator $cafe2athena$ as follows:

Datatype modules:

```
cafe2athena(mod! M1 {[m1] ...})
=
datatype m1 cafe2athena(...)

cafe2athena(mod* M2 {[m2] ...})
=
domain m2 cafe2athena(...)
```

OTS modules (sort representing state space, initial state, transitions):

```
cafe2athena(mod S {*[Sys]* op init : → Sys .
bop a : Sys V_{j_1} ... V_{j_n} → Sys ...})
=
structure Sys := init | (a Sys V_{j_1} ... V_{j_n})
```

Observers:

```
cafe2athena(bop o : Sys V_i1 ... V_in → V)
=
declare o := [Sys V_i1 ... V_in] → V
```

Variables:

```
cafe2athena(Var x_i1 : V_i1 ... Var x_in : V_in)
=
define [x_i1 ... x_in] := [?V_i1 ... ?V_in]
```

Init axiom:

```
cafe2athena(eq o(init, x_i1, ..., x_in) = f(x_i1, ..., x_in))
=
assert* init-axiom := ((o init x_i1 ... x_in) = f x_i1 ... x_in)
```

where x_{i1},...,x_{in} are V_{i1} ...V_{in} sorted CafeOBJ variables and $f(x_{i1}, ..., x_{in})$ is the value of the observer at the initial state.

Effective condition:

```
cafe2athena(op c-a : Sys V_j1 ... V_jn → Bool .
eq c-a(w, x_j1, ..., x_jn) = g(w, x_j1, ..., x_jn) .)
=
define c-a := lambda (w x_j1 ... x_jn) (g w x_j1 ... x_jn)
```

where w is a hidden sorted variable and $x_{j1}, ..., x_{jn}$ are V_{j1}...V_{jn} sorted CafeOBJ variables.

Transition axioms:

```
cafe2athena(eq o(a(w, x_j1, ..., x_jn), x_i1, ..., x_in) = e-a(w, x_j1, ..., x_jn, x_i1, ..., x_in)
        if c-a(w, x_j1, ..., x_jn) .)
=
assert a-axiom :=
((o(a w x_j1 ...x_jn) x_i1 ... x_in) = (e-a w x_j1 ... x_jn x_i1 ... x_in)
        if (c-a w x_j1 ... x_jn))
```

where e-$a(w, x_{j1}, ..., x_{jn}, x_{i1}, ..., x_{in})$ is the changed value of the observer after the application of the transition.

Here we present the declaration of an invariant property in CafeOBJ terms and the definition of the induction schema.

```
-- declaration of the invariant property
mod INV {
-- arbitrary values
op s : -> Sys .
ops i j : -> Pid .
-- name of invariant to prove
op inv1 : Sys Pid Pid -> Bool
-- CafeOBJ variables
var S : Sys
vars I J : Pid
```

```
-- invariant to prove
eq inv1(S,I,J) = (pc(S,I) = cs and pc(S,J) = cs implies I = J) . }

-- declaration of the inductive step
mod ISTEP {
pr(INV)
-- arbitrary values
op s' : -> Sys
-- name of formula to prove in each induction case
op istep1 : -> Bool
-- formula to prove in each induction case
eq istep1 = inv1(s,i,j) implies inv1(s',i,j) .  }
```

The definition of the corresponding invariant in Athena can be seen below (the induction schema is pre defined in Athena).

```
define (inv1 s) := (forall ?i ?j . pc s ?i = cs & pc s ?j = cs
==> ?i = ?j)
```

Here we present the proof score of the desired invariant property in CafeOBJ, for the initial state and when the transition $try(s, k)$ is applied.

```
-- proof score
-- I. Base case
open INV .
  red inv1(init,i,j) .
close .

-- II. Induction case
-- 1. try(s,k)
open ISTEP  .
-- arbitrary values
  op k : -> Pid .
-- successor state
  eq s' = try(s,k) .
-- check
  red istep1 .
close
```

The corresponding proof skeleton in Athena can be seen below.

```
by-induction (forall ?s . goal-property ?s) {
        init => (!prove (goal-property init))
        | (state as (try s k)) =>
        pick-any i:Pid j:Pid
        assume hyp := (state at i = cs & state at j = cs) {
                        goal := (i = j);
                        goal from (ab)
                }
        }
```

5.2.2 Semantic correctness of the translation

We should mention at this point that we restricted the translation to non-parametric modules, and that the only hidden sort of a specification is the

hidden sort denoting the state space of the OTS. By enforcing these two restrictions on the style of the OTS/CafeOBJ specifications it is trivial to see that the semantics for both languages are equivalent. Indeed the parametric module system does not add to the expressiveness of the language, however not supporting it will result in an overhead in the specification code. Second, the second assumption does not impose any violation of generality because we are only interested in translating OTS specifications and not behavioral specifications in general.

Thus it is safe to claim the semantic correctness of the translation since in both languages the underlying semantics is basically the same, i.e. Order-sorted Conditional Equational Logic (in which constructors are (explicitly) used). However in order to be more accurate we also here that the translation is complete w.r.t. invariant properties.

Lemma 5.1: Let SP be an OTS/CafeOBJ specification and SP' an OTS/Athena specification obtained by the proposed translation. Also assume that:

$o\ (a\ w\ x_{j1}\ ...x_{jn})\ x_{i1}\ ...\ x_{in} = v_{oi}$ after the application of the transition a in SP',

then

$o(a(w, x_{j1}, ..., x_{jn}), x_{i1}, ..., x_{in}) = v_{oi}$ after the application of the corresponding transition a in SP.

To prove this, we show that a conditional transition $ceq\ o(a(w, x_{j1}, ..., x_{jn}), x_{i1}, ..., x_{in}) = rv_{oi}\ if\ c\text{-}a(w, x_{j1}, ..., x_{jn})$ in SP can be applied to obtain the equation, that means that we show $c\text{-}a(w, x_{j1}, ..., x_{jn})$ is true and also that $rv_{oi} = v_{oi}$.

From the assumption, the condition of the transition rule a in SP' holds, that is: $(c\text{-}a\ w\ x_{j1}\ ...\ x_{jn}) = true$. By applying the operator $cafe2athena$ inversely in this effective condition, we obtain that: $cafe2athena^{-1}(c\text{-}a\ w\ x_{j1}\ ...\ x_{jn}) = c\text{-}a(w, x_{j1}, ..., x_{jn}) =$ true.

This means that the effective condition of the corresponding transition a in SP also holds and thus the transition will be effectively applied, i.e. $ceq\ o(a(w, x_{j1}, ..., x_{jn}), x_{i1}, ..., x_{in}) = e\text{-}a(w, x_{j1}, ..., x_{jn}, x_{i1}, ..., x_{in})\ if\ c\text{-}a(w, x_{j1}, \ldots, x_{jn})$.

By applying again the operator $cafe2athena$ inversely to the returned value of the observer, we obtain: $cafe2athena^{-1}(e\text{-}a(w, x_{j1}, ..., x_{jn}, x_{i1}, ..., x_{in}) = (e\text{-}a\ w\ x_{j1}\ ...\ x_{jn}\ x_{i1}\ ...\ x_{in}) = v_{oi}$. ■

This means that the corresponding observers in SP and SP' after the application of the corresponding transitions, return the same values.

The following theorem is about a kind of completeness of our translation system. The theorem states that if a state property is invariant of an OTS/ CafeOBJ specification, then the translated property is also an invariant of the translated OTS/Athena specification.

Theorem 5.2: Let SP be an OTS/CafeOBJ specification and P a state property in SP. Assume SP' and P' are translated from SP and P. Then, $SP' \models Inv'_P$ whenever $SP \models Inv_P$.

Proof. Assume $SP' \not\models Inv'_P$. This means that there exists a counter-example state in SP', say s, reachable from the initial state such that $P'(s) = false$. Assume that in order to reach this counter-example state the following sequence of transition rules is applied: $a_{i0}(s_o), a_{i1}(s_1), ..., a_{ik}(s_k)$.

In SP respectively, after the application of the corresponding transitions in the corresponding initial state, we obtain a state, say w. By applying lemma 3.1 for each of the above transition rules, we see that the corresponding observers in this state have the same values with those in SP' (a).

In [94] authors present two ways to go from equational logic to first-order logic. More precisely, they state that many examples for comorphisms arise from subinstitutions. For example, many-sorted equational logic is subinstitution of many-sorted first-order logic (b).

In addition, in [95] authors state that the institution morphism $\mu = (\Phi, \alpha, \beta)$ from FOL to EQL maps any FOL signature (S, F, P) to the corresponding algebraic one (S, F), regards any set of equations as a set of first-order sentences over $(S, F, 0)$, and regards any (S, F, P)-model as a (S, F)-algebra by forgetting the interpretations of predicate names in P. Also, it is easy to show that the satisfaction property holds (b).

From (a) and (b), we can conclude that in w the corresponding state property will not hold, i.e. $P(w) = false$, and $SP \not\models Inv_P$, which contradicts our hypothesis ■

5.2.3 Cafe2Athena Tool

In order to make the proposed methodology more agile, we have developed a tool that takes as input an OTS specification written in CafeOBJ and automatically produces an Athena specification, implementing the rules of translation we previously presented. The tool is written in Java and hides the details of the translation, since the users load a CafeOBJ specification file, press the "Translate to Athena" button and get the corresponding Athena specification. Having obtained the Athena specification of a system we can give it directly to the Athena prover and adopt the proposed methodology. More information about the *Cafe2Athena* tool can be found in [48, 96].

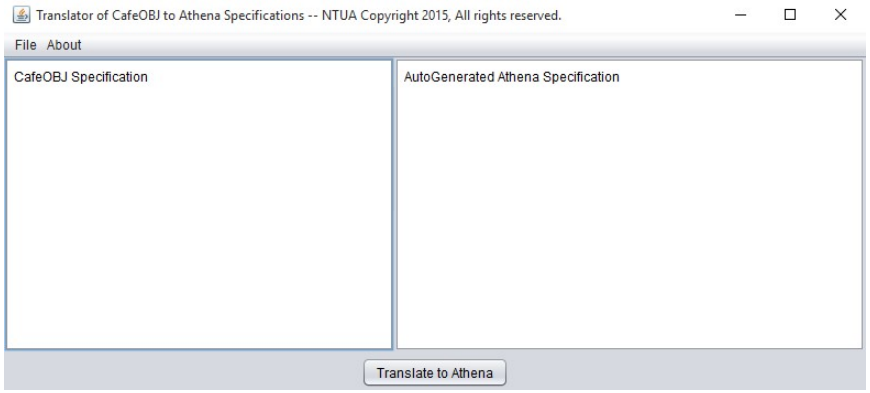


Figure 13: A snapshot of tool that translates OTS/CafeOBJ specifications into Athena specifications

To demonstrate the benefits and the effectiveness of the proposed methodology we present its application to appropriate case studies.

5.3 A Mutual Exclusion Protocol Using an Atomic Instruction

In our mutex system, we have a set of processes, each of which is executing code. A process, at any system state, is either in some critical section of the code or in some remainder (waiting) section. When a process p enters its critical section, the resulting state becomes *locked*. When p exits the critical section, the resulting state is unlocked. For p to enter its critical section in some state s, p must be *enabled* in s. A process p is *enabled* in s iff p is in its remainder section in s and s is not locked. This is, therefore, the effective condition of the *enter* state transition for a given process. The effective condition of the *exit* transition is for the process to be in its critical section. We have two observer functions, one that takes a state s and a process id p and tells us what section of the code p is executing in s (critical or remainder), and a function that takes a state s and tells us whether s is locked.

5.3.1 Step 1. Specification in CafeOBJ.

The specification of the mutex OTS in CafeOBJ is presented below.

```
mod! LABEL {
  [Label]
  ops rs cs : -> Label
  op _=_ : Label Label -> Bool {comm}
  var L : Label
  eq (L = L) = true .
  eq (rs = cs) = false .}
```

```
mod* PID {
  [Pid]
  op _=_ : Pid Pid -> Bool {comm}
  var I : Pid
  eq (I = I) = true .}

mod* MUTEX {
  pr(LABEL + PID)
  *[Sys]*
  -- an arbitrary initial state
  op init : -> Sys
  -- observation functions
  bop at : Sys Pid -> Label
  bop locked : Sys -> Bool
  -- transition functions
  bop enter : Sys Pid -> Sys
  bop exit : Sys Pid -> Sys
  -- CafeOBJ variables
  var S : Sys
  vars I J : Pid
  -- init
  eq at(init,I) = rs .
  eq locked(init) = false .
  -- enter
  op c-enter : Sys Pid -> Bool
  eq c-enter(S,I) = ((at(S,I) = rs) and not locked(S)) .
  ceq at(enter(S,I),J) = cs if (I = J) and c-enter(S,I) .
  ceq at(enter(S,I),J) = at(S,J) if not((I = J) and c-enter(S,I)) .
  ceq locked(enter(S,I)) = true if c-enter(S,I) .
  ceq enter(S,I) = S if not c-enter(S,I) .
  -- exit
  op c-exit : Sys Pid -> Bool
  eq c-exit(S,I) = (at(S,I) = cs) .
  ceq at(exit(S,I),J) = rs if (I = J) and c-exit(S,I) .
  ceq at(exit(S,I),J) = at(S,J) if not((I = J) and c-exit(S,I)) .
  ceq at(exit(S,I),J) = at(S,J) if not(I = J) and not c-exit(S,I) .
  ceq locked(exit(S,I)) = false if c-exit(S,I) .
  ceq exit(S,I) = S if not c-exit(S,I) .}
```

5.3.2 Step 2. Specification in Athena.

Using the Cafe2Athena tool we obtain the specification of the mutex OTS in Athena. Some parts of the specification are explained below.

A domain of process identifiers (*Pid*) and a datatype for code labels (*cs* and *rs*, for critical and remainder section, respectively) have been introduced:

```
module Mutex {
domain Pid
datatype Label := rs | cs
assert label-axioms := (datatype-axioms "Label")
```

Here, *rs* and *cs* are the (nullary) *constructors* of the *Label* algebra. The datatype-axioms of *Label* are quantified sentences that assert no-confusion and no-junk conditions for the constructors. The effect of the *assert* command is to insert those conditions into the global *assumption base*.

States are formalized as a structure and the state transitions as constructors of this structure, as follows:

```
structure State := init | (enter Pid State)
| (exit Pid State)
```

In the following we can see the declaration of the observer functions:

```
declare at: [Pid State] -> Label
declare locked: [State] -> Boolean
declare at: [Pid State] -> Label
declare locked: [State] -> Boolean
```

The "*at*" function tells us the label of a given process in a given state. Binary function symbols can be used in infix in Athena (the default notation is prefix), so if *p* and *s* are terms of sort *Pid* and *State*, respectively, then *(p at s)* gives us the label of *p* in state *s*. The term *(locked s)* tells us whether or not *s* is locked.

We now present the axioms that define the initial state, as well as the pre- and post-conditions of the two state transitions (entering and exiting). First, appropriate variables for the given sorts are defined.

```
define [i j s s'] := [?i:Process ?j:Process ?s:State
?s':State]

assert* init-axioms := [(_ at init = rs)
                        (~ locked init)]
```

The initial-state axioms are simple enough: every process in the initial state is in the remainder section, and the initial state is not locked.

For the transition axioms of *enter* we see it is helpful to define the effective condition as a separate procedure, called *enabled-at*:

```
define (enabled-at i s) := (s at i = rs & ~ locked s)
```

The axioms for the *enter* transition for example, are presented below:

```
assert* enter-axioms :=
 [(i enabled-at s ==> locked i enter s)
  (i enabled-at s ==> i at i enter s = cs)
  (i enabled-at s & j =/= i ==> j at i enter s = j at s)
  (~ i enabled-at s ==> i enter s = s)]
```

Note that *assert** automatically universally quantifies all free variables, so, for instance, the first of the four *enter-axioms*, when fully written out in prefix form, is this:

```
(forall ?i:Mutex.Pid
  (forall ?s:Mutex.State
    (if (and (= (Mutex.at ?i:Mutex.Pid ?s:Mutex.State)
                Mutex.rs)
             (not (Mutex.locked ?s:Mutex.State)))
        (Mutex.locked (Mutex.enter ?i:Mutex.Pid
                                   ?s:Mutex.State)))))
```

Finally, we have the axioms for the *exit* transition:

```
assert* exit-axioms :=
 [(i at s = cs ==> i at i exit s = cs)
  (i at s = cs & j =/= i ==> j at i enter s = j at s)
  (i at s = cs ==> ~ locked i exit s)
  (i at s =/= cs ==> i exit s = s)]
```

5.3.3 Step 3. Define the desired goal and falsify it with Athena.

The desired mutual exclusion property satisfied by the algorithm is that there is at most one process in the critical section at any given moment. This property can be rephrased as "*if there are two processes in the critical section, then those processes are identical*".

In Athena, the desired goal is defined as follows:

```
define (goal-property s) :=
    (forall i j . i at s = cs & j at s = cs ==> i = j)

define goal := (forall s . goal-property s)
```

Let us see if we can falsify the goal by examining 100 states:

```
> (falsify  goal 100)

List: ['success
|{
?i:Pid := Pid_1
?j:Pid := Pid_2
?s:State := (enter Pid_2
       (exit Pid_1
             (enter Pid_1 init)))
}|]

>
```

Since Athena returned the above argument, it means that either the goal does not hold, or there is something wrong with the specification. Since we know which transitions cause the falsification of the goal, we can use the simulate method to see the value of the observers in this problematic state.

In our case, calling *simulate (p2 enter p1 exit p1 enter init)*, results in the following output:

```
State after (p1 enter init):

locked:                                    p1 at:
------------                               -------------------
true                                       cs

State after (p1 exit (p1 enter init)):

locked:                                    p1 at:
------------                               -------------------
false                                     cs

State after (p2 enter (p1 exit (p1 enter init))):

locked:                                    p2 at:
------------                               -------------------
true                                       cs
```

If we observe the values returned by the observers in this state we see that while p2 exits the critical section it basically remains in the critical section. Thus we understand that we there must exist an error in the definition of the exit axioms (in particular, the first axiom was miswritten as $(i\ at\ s = cs\ \Longrightarrow\ i\ at\ i\ exit\ s = cs)$ instead of $(i\ at\ s = cs\ \Longrightarrow\ i\ at\ i\ exit\ s = rs)$).

After redefining the axiom, we falsify again the desired goal, and Athena returns the term *failure*. Thus we proceed to the next step of our methodology.

5.3.4 Step 4. Start the proof of the desired goal using the Proof Scores methodology.

We continue with the proof score approach in CafeOBJ, until we reach the following case, in which CafeOBJ returns *false*:

```
open ISTEP .
-- arbitrary values
op k : -> Pid .
-- assumptions
-- eq c-enter(s,k) = true .
eq at(s,k) = rs .
eq locked(s) = false .
eq i = k .
eq (j = k) = false .
eq at(s,j) = cs .
-- successor state
eq s' = enter(s,k) .
red istep1 .
close
```

To discard this case we must use a lemma. Based on the equations that define this state, a possible lemma is the following: *pc(S,J) = cs implies*

locked(S) = true. To test if this lemma actually discards the problematic case we use the following proof score, and CafeOBJ returns *true*.

```
open ISTEP .
-- arbitrary values
op k : -> Pid .
-- assumptions
-- eq c-enter(s,k) = true .
eq at(s,k) = rs .
eq locked(s) = false .
eq i = k .
eq (j = k) = false .
eq at(s,j) = cs .
-- successor state
eq s' = enter(s,k) .
-- red istep1 .
red inv2(s,i,j) implies istep1 .
close
```

5.3.5 Step 5. Falsify the discovered lemma with Athena.

Next, after defining the lemma in Athena we attempt to falsify it using the following code:

```
define (lemmal-property s) :=
(forall i j . j at s = cs ==> locked s)

define lemma := (forall s . lemma-property s)

> (falsify lemma 100)

Term: 'failure
```

Since Athena returns the term *failure* we continue the proof score using this lemma.

5.3.6 Step 6. Continue the proof with the proof scores approach.

Checking the rest of the cases in CafeOBJ will result in the completion of the proof score of the desired property. It is interesting to point out that in our example, during the proof score of the lemma the original property under verification was required as part of the inductive hypothesis. This case is shown below.

```
open ISTEP .
-- arbitrary values
op k : -> Pid .
-- assumptions
-- eq c-exit(s,k) = true .
eq at(s,k) = cs .
eq (j = k) = false .
```

```
eq at(s,j) = cs .
-- successor state
eq s' = exit(s,k) .
red inv1(s,j,k) implies istep2 .
close
```

5.3.7 Step 7. Create an Athena proof based on the gained insights.

The above pattern in a proof score denotes a situation where simultaneous induction must performed. Together with the case splits and lemmas used, such information is essential to the construction of the Athena proof. Also, by taking a closer look at the invariant and the lemma used ($(i\ at\ s = cs\ \&\ j\ at\ s = cs \implies i = j$ *and* $j\ at\ s = cs \implies locked\ s$, respectively) it is not difficult to understand that a strengthened goal can be formulated out of them: $i\ at\ s = cs\ \&\ j\ at\ s = cs \implies i = j\ \&\ locked\ s$, which we will verify in Athena. The new goal is defined as follows:

```
define (new-goal-property s) :=
(forall i j . i at s = cs & j at s = cs ==> i = j & locked s)

define new-goal := (forall s . new-goal-property s)
```

This strengthened goal is automatically proved, as the following shows:

```
> (!state-induction new)

Theorem: (forall ?s:State
           (forall ?i:Pid
             (forall ?j:Pid
               (if (and (= (at ?i:Pid ?s:State)
                           cs)
                        (= (at ?j:Pid ?s:State)
                           cs))
                   (and (= ?i:Pid ?j:Pid)
                        (locked ?s:State))))))
```

Part of the detailed structured Athena proof of the (strengthened) goal for our example is shown below.

Theorem 5.3. *For all states s' and processes i and j, if i and j are in their critical sections in s', then $i = j$ and s' is locked.*

Proof. By structural induction on s'. When s' is the initial state the result is trivial because the antecedent is false, as all processes are in their remainder sections initially. Suppose now that s' is of the form *(k enter s)*. Pick any processes i and j and assume both are in their critical sections in s'. We then

need to show that $i = j$ and that $s' = (k\ enter\ s)$ is locked. The inductive hypothesis here is:

$$i\ at\ s = cs\ \&\ j\ at\ s = cs \Longrightarrow i = j\ \&\ locked\ s \tag{1}$$

We distinguish two cases:

1. *Case 1*: k is enabled at s. Then $(k\ at\ s' = cs)$ and $(locked\ s')$ follow from the *enter* axioms. Thus, we only need to show $i = j$. By contradiction, suppose that $i \neq j$. Then either $i \neq k$ or $j \neq k$. [9] So assume first that $i \neq k$ (the reasoning for the case $j \neq k$ is symmetric). Then, from the *enter* axioms and the assumption that k is enabled at s, we conclude $i\ at\ s' = i\ at\ s$, hence $i\ at\ s = cs$. Now applying the inductive hypothesis to the above assumption, we conclude $(locked\ s)$. However, that contradicts the assumption that k is enabled at s, as that assumption means that s is *not* locked.

2. *Case 2*: k is not enabled at s. In that case, by the *enter* axioms, we get

   $$(k\ enter\ s = s)$$

   i.e., $s' = s$, and the result now follows directly from the inductive hypothesis.

Finally, suppose that s' is of the form $(k\ exit\ s)$. Again pick any processes i and j and assume both are in their critical sections in s'. We again need to show that $i = j$ and that $s' = (k\ exit\ s)$ is locked. The inductive hypothesis here is the same as before, (1). We distinguish two cases again, depending on whether or not the effective condition of the *exit* transition holds:

1. *Case 1*: $(k\ at\ s = cs)$. We proceed by contradiction. First, by applying the inductive hypothesis to the conjunction of $(k\ at\ s = cs)$ with itself, we obtain $(locked\ s)$. Also, by the *exit* axioms, we get

   $$k\ at\ s' = (k\ exit\ s) = rs$$

   i.e.,

   $$k\ at\ s' = rs. \tag{2}$$

   The *exit* axioms also imply that s' is *not* locked. We can now conclude that

   $$i \neq k \tag{3}$$

   because otherwise, if $i = k$, the assumption that i is in cs in state s' would contradict (2). Hence, by the *exit* axioms, we get

   $$i\ at\ s' = i\ at\ s. \tag{4}$$

[9] Clearly, if neither of these hold, i.e., if $i = k$ and $j = k$, then we could also have $i = j$, contradicting our hypothesis.

Therefore, from (4) and the assumption that i is in cs in s', we get i *at* s $= cs$. But now applying the inductive hypothesis to i *at* $s = cs$ and to *(*k *at* $s = cs$*)* yields $i = k$, contradicting (3).

2. *Case 2*: *(*k *at* $s \neq cs$*)*. In that case the *exit* axioms give *(*k *exit* s $= s$, i.e., $s' = s$, and the result follows directly from the inductive hypothesis.

□

The above informal proof can be formulated in Athena at the same level of abstraction and with the exact same structure. Moreover, the proof colloquialism "the reasoning for that case is symmetric" that appears in the *enter* transition can be directly accommodated by abstracting the symmetric reasoning into a *method* and then applying that method to multiple instances. Likewise, the treatment of *enter* and *exit* is symmetric when their effective conditions are violated, in which case the result follows directly from the inductive hypothesis, and this commonality too can be easily factored out into a general method. The entire proof, along with these two methods, can be seen below. Note that the proof doesn't use external theorem provers. Instead, it uses Athena's own library *chain* method, which allows for limited proof search. The *chain* method extends the readability benefits of equational chains into arbitrary implication chains.

```
# This is the ``symmetric'' reasoning that appears in two
# places in the treatment of enter. We abstract it here
# into one single generic method M.

define (M inequality enabled-premise IH) :=
  match [inequality enabled-premise] {
    [(~ (i = k)) (((k at s) = rs) & (~ (locked s)))] =>
      (!chain->
       [inequality
    ==> (enabled-premise & inequality) [augment]
    ==> (i at k enter s = i at s)      [enter-axioms]
    ==> (i at s = i at k enter s)      [sym]
    ==> (i at s = cs)                  [(i at k enter s = cs)]
    ==> (i at s = cs & i at s = cs)    [augment]
    ==> (locked s)                     [IH]
    ==> (locked s & ~ locked s)        [augment]
    ==> false                          [prop-taut]])
  }

# This method handles all cases where the effective condition
# is violated.

define (direct-ih hyp s failed-ec IH transition-axioms) :=
 match hyp {
   (((i at s') = cs) & ((j at s') = cs)) =>
```

```
    let {s=s' := (!chain->
                   [failed-ec ==> (s' = s) [transition-
                                            axioms]])}
      (!chain-> [hyp ==> (i at s = cs & j at s = cs) [s=s']
                     ==> (i = j & locked s)          [IH]
                     ==> (i = j & locked s')         [s=s']])
  }

# The main proof:

by-induction (forall s . gp s) {
  init => (!spf (gp init) (ab))
| (s' as (k enter s)) =>
   pick-any i:Pid j:Pid
    let {IH := (forall i j . i at s = cs & j at s = cs
                          ==> i = j & locked s)}
    assume hyp := (i at s' = cs & j at s' = cs)
     conclude goal := (i = j & locked s')
      (!two-cases
       assume case1 := (k enabled-at s)
        let {s'-locked := (!chain-> [case1
                             ==> (locked s') [enter-axioms]]);
             s-not-locked := (!chain-> [case1
                             ==> (~ locked s) [right-and]]);
             i=j := (!by-contradiction (i = j)
                      assume h := (i =/= j)
                       let {D := {(i =/= k | j =/= k) from h}}
                        (!cases D
                          assume i=/=k := (i =/= k)
                           (!M i=/=k case1 IH)
                          assume j=/=k := (j =/= k)
                           (!M j=/=k case1 IH)))}
        (!both i=j s'-locked)
       assume case2 := (~ k enabled-at s)
        (!direct-ih hyp s case2 IH enter-axioms))
| (s' as (k exit s)) =>
  pick-any i:Pid j:Pid
   let {IH := (forall i j . i at s = cs & j at s = cs
                         ==> i = j & locked s)}
    assume hyp := (i at s' = cs & j at s' = cs)
     conclude goal := (i = j & locked s')
      (!two-cases
       assume case1 := (k at s = cs)
        (!by-contradiction goal
         assume -goal := (~ goal)
          let {locked-s :=
                (!chain-> [case1
                        ==> (case1 & case1) [augment]
                        ==> (locked s)      [IH]]);
               p2 := (!chain-> [case1
                             ==> (k at s' = rs) [exit-axioms]]);
               i=/=k :=
                 (!by-contradiction (i =/= k)
                  assume h := (i = k)
```

```
                    (!chain-> [p2
                          ==> (i at s' = rs) [h]
                          ==> (cs = rs)      [(i at s' = cs)]
                          ==> (cs = rs & cs =/= rs)  [augment]
                          ==> false            [prop-taut]]))}
          (!chain-> [i=/=k
                ==> (i at s' = i at s)  [exit-axioms]
                ==> (i at s = cs)       [(i at s' = cs)]
                ==> (i at s = cs & k at s = cs) [augment]
                ==> (i = k)                     [IH]
                ==> (i = k & i =/= k)           [augment]
                ==> false                       [prop-taut]]))
       assume case2 := (k at s =/= cs)
        (!direct-ih hyp s case2 IH exit-axioms))
}
```

5.4 Alternating Bit Protocol (ABP)

Alternating Bit Protocol is a communication protocol that provides sending reliable messages on unreliable channels. It is often used as a case study, especially for tools dedicated to the verification of complex systems. Even if the protocol seems to be simple, its complete algebraic specification is complex and its formal proof is large. We show that using our methodology, we can better understand the specified system and also some parts of the proving process can be automated.

We describe briefly the protocol. Two processes, *Sender* and *Receiver*, that do not share any common memory use two channels to communicate with each other. Sender sends repeatedly a pair $< bit1, pac >$ of a bit and a packet to the Receiver over one of the channels, let's say $channel1$. When Sender gets a bit from Receiver over the other channel, let's say $channel2$, if it does not equal $bit1$, Sender selects the next packet for sending and alternates $bit1$. Receiver puts $bit2$ into $channel2$ repeatedly. When Receiver gets a pair $< b, p >$ such that b is the same with $bit2$, it stores p into a list and alternates $bit2$.

Initially both channels are empty and the Sender's bit is the same with the Receiver's bit. We assume that the channels are unreliable, meaning that the data in the channels may be lost and/or duplicated, but not exchanged or damaged. The packets sent by Sender to Receiver through channel1 are indexed by the natural numbers and are of the form $pac(0), pac(s0), pac(s^n 0)$, where $pac : Nat \rightarrow Packet$ is the constructor for packets. The bits sent by both Sender and Receiver into the communication channels are modelled by the boolean values true and false. The communication channels and the packets received by the Receiver are modelled by queues [97].

(a) channel1 consists of queues of pairs of bits and packets of the form $< b_1, p_1 >, \ldots, < b_n, p_n >$.

(b) channel2 consists of queues of bits of the form $b_1, \ldots, b_n$.

(c) The list of packets received by Receiver consists of packets of the form $p_1, \ldots, p_n$.

Here we do not present in details the steps of the proposed methodology, as before, due to the complexity of the Abp protocol. We briefly present some parts of the application of the proposed methodology in the case study.

The definitions of the state space, the transitions and the observers in Athena are presented below.

```
structure Sys := init|(send1 Sys)|(rec1 Sys)|(send2 Sys)
|(rec2 Sys)|(drop1 Sys)|(dup1 Sys)|(drop2 Sys)|(dup2 Sys)

declare fifo1: [Sys] -> PFifo
declare fifo2: [Sys] -> BFifo
declare next: [Sys] -> Nat
declare list: [Sys] -> List
declare bit1: [Sys] -> Boolean
declare bit2: [Sys] -> Boolean
```

Transitions *send1* and *send2* model Sender's sending bits & packets and Receiver's sending bits, respectively. Transitions *rec1* and *rec2* represent Sender's receiving bits and Receiver's receiving pairs of bits & packets, respectively.

We suppose that any data in a channel can be lost and/or duplicated. But since only the top data in a channel is extracted, it suffices that the effects of losing and duplicating data can be seen when the data becomes top in the channel. Thus, *drop1* represents the action of dropping the first data of channel1, while *drop2* represents the same for channel2. Finally, transitions *dup1* and *dup2* model the actions of duplicating the first data of channel1 and channel2, respectively.

The functions *fifo1* and *fifo2* represent channel1 and channel2, respectively. The ordinal of the packet sent next by the Sender is modelled by the function *next* and the packets received by the Receiver are stored in the *list*. Finally, *bit1* corresponds to the Sender's bit and *bit2* to the Receiver's.

The initial-state axioms are the following: the channels in the initial state are empty, the packet to be sent next by the Sender is indexed by the number zero, the list of the received packets does not contain any packet and the bits of Sender and Receiver are the same.

```
assert* init-axioms :=
[((fifo1 init) = empty)((fifo2 init) = empty)
((next init) = zero)((list init) = nil)
((bit1 init) = false)((bit2 init) = false))]
```

The axioms for the *rec1* transition for example, are presented below:

```
assert* rec2-axioms :=
[((fifo1 (rec2 S)) = (get (fifo1 S)) if (c-rec2 S))
((fifo2 (rec2 S)) = (fifo2 S))
((bit1 (rec2 S)) = (bit1 S))
```

```
((bit2 (rec2 S)) <==> (not (fst (top1 (fifo1 S)))) if
(((bit2 S) = (fst (top1 (fifo1 S)))) and (c-rec2 S)))
((bit2 (rec2 S)) = (bit2 S) if ((not ((bit2 S) =
(fst (top1 (fifo1 S))))) and (c-rec2 S)))
((next (rec2 S)) = (next S))
((list (rec2 S)) = (++ (snd (top1 (fifo1 S)))(list S)) if
(((bit2 S) = (fst (top1 (fifo1 S)))) and (c-rec2 S)))
((list (rec2 S)) = (list S) if ((not ((bit2 S) = (fst (top1
(fifo1 S))))) and (c-rec2 S)))
((rec2 S) = S if (not c-rec2 S))]
```

When the Receiver receives a pair of bit & packet from the Sender (through channel1) the following things happen; the first element from channel1 is removed, *channel2*, *bit1* and *next* stay the same, bit2 changes (if it was true it becomes false and the opposite) if it is the same with bit1 and the effective condition of the transition holds (bit2 stays the same if it is not the same with bit1 and the effective condition of the transition holds), the packet sent by the Sender is stored in the *list* if bit2 is the same with bit1 and the effective condition of the transition holds (*list* stays the same if bit2 is not the same with bit1 and the effective condition of the transition holds), and finally, the state does not change if the effective condition of the transition does not hold.

One desired property satisfied by the communication protocol, is the following; when Receiver receives the n^{th} packet:

- Receiver has received the $n+1$ packets $p_0, \ldots, p_n$ in this order,
- each p_i for $i = 0, \ldots, n$ has been received only once, and
- no other packets have been received.

The property is called the reliable communication property in the literature. The definition of this property in Athena terms is shown in the box below[10]:

```
define (inv1 s) :=
((bit1 s = bit2 s  ==> mk next s = packet next s ++ list s) &
(bit1 s =/= bit2 s  ==> mk next s = list s))
```

Following the proposed methodology presented before, and after getting all the insightful information about the system using the Proof Scores approach we verified the desired goal in Athena.

Some interesting points of the proof we would like to present are the following: In order to prove the desired goal, ten lemmas were used which were discovered using the proof scores approach. In order to prove the required lemmas we used simultaneous induction (another insight gained by the proof scores approach). The lemmas and their proof schema are shown below.

[10]Where $mk(n) = pac(n) \ldots pac(0)$.

```
define (inv2 s) :=
((non-empty channel2 s) ==> bit1 s = top channel2 s | bit2 s
= top channel2 s)

define (inv3 s) :=
(non-empty channel1 s & bit2 s = fst top channel1 s ==> bit1 s
= fst top channel1 s & packet next s = snd top channel1 s)

define (inv4 s) :=
(forall ?bit . non-empty channel2 s & bit1 s =/= top channel2 s
& ?bit in channel2 s ==> top channel2 s = ?bit)

define (inv5 s) :=
(forall ?bit . non-empty channel2 s & ?bit in channel2 s & bit1
s =/= ?bit ==> bit2 s = ?bit)

define (inv6 s) :=
(forall ?pair . non-empty channel1 s & bit2 s = fst top channel1
s & ?pair in channel1 s ==> top channel1 s = ?pair)

define (inv7 s) :=
(forall ?pair . non-empty channel1 s & ?pair in channel1 s
& bit2 s = fst ?pair ==> bit1 s = fst ?pair & packet next s
= snd ?pair)

define (inv8 s) :=
(forall ?bfifo1 ?bfifo2 ?bit1 ?bit2 ?bit3 .
channel2 s = ?bfifo1  ?bit1 ++ ?bit2 ++ ?bfifo2 & not ?bit1
= ?bit2 ==> ((?bit3 in ?bfifo2 ==> ?bit2 = ?bit3) & ?bit2
= bit2 s))

define (inv9 s) :=
(forall ?pfifo1 ?pfifo2 ?pair1 ?pair2 ?pair3 .
channel1 s = ?pfifo1  ?pair1 ++ (?pair2 ++ ?pfifo2) & ?pair1
=/= ?pair2 ==> ((?pair3 in ?pfifo2 ==> ?pair2 = ?pair3) &
?pair2 = bit1 s
with packet next s))

define (inv10 s) :=
(forall ?bit . bit1 s = bit2 s ==> ?bit in channel2 s ==>
?bit = bit2 s)

define (inv11 s) :=
(forall ?pair . bit1 s =/= bit2 s ==> ?pair in channel1 s ==>
?pair = bit1 s with packet next s)
```

```
ABP |- {inv2 & inv3 & inv4 & inv5 & inv6 & inv7 & inv9 & inv10
& inv11 & inv8}

ABP + {inv2 & inv3 & inv4 & inv5 & inv6 & inv7 & inv9 & inv10
```

```
& inv11 & inv8} |- inv1
```

During the proof of the initial goal the most difficult parts were transitions *rec2* and *rec1* whereas in the proof of the ten lemmas the most difficult cases were those of transitions *rec2* and *send1*.

Part of the detailed structured Athena proof of the desired goal is presented below.

```
by-induction (forall ?s . inv1 ?s) {
 init => (!prove (inv1 init))
| (state as (rec1 s)) =>
        let {
        part1 := conclude (bit1 state = bit2 state
        ==> mk next state = packet next state ++ list
        state)

        assume hyp1 := (bit1 state = bit2 state)
        {
        goal1a := (mk next state = packet next
        state ++ list state);
        case1 := assume case1 := (channel2 s = empty) {
                goal1a from (ab)
        };
        case2 := assume case2 := (not channel2 s = empty) {
                goal1a from (ab)
        };
        goal1a from case1, case2
        };
        part2 := conclude (bit1 state =/= bit2 state
        ==> mk next state = list state)
        assume hyp2 := (bit1 state =/= bit2 state) {
        goal1b := (mk next state = list state);
        case1 := assume case1 := (channel2 s = empty) {
                goal1b from (ab)
        };
        case2 := assume case2 := (not channel2 s = empty) {
        p1 := (exists ?b ?bs . channel2 s = ?b ++ ?bs) from
        (ab);
        pick-witnesses x y for p1 {
                case2a := assume case2a := (bit1 s = x) {
                        goal1b from (ab)
                };
                case2b := assume case2b := (not bit1 s = x) {
                        case2b-1 := assume case2b-1 :=
                        (bit2 s = x) {
                                goal1b from (ab)
                        };
                        case2b-2 := assume case2b-2 :=
                        (not bit2 s = x) {
                                goal1b from (ab)
                        };
                        goal1b from case2b-1, case2b-2
                };
```

```
                goal1b from case2a, case2b
                } };
        goal1b from case1, case2
        } }
(!derive (inv1 state) [part1 part2])
| (state as (rec2 s)) =>
        ...
```

Finally, in order to check if any part of the proof can be done automatically, we defined another ATP method (using SPASS), called "*do*", which takes as input the goal you're trying to prove along with the state-transition operator it involves. Roughly speaking, this method knows the effective condition and transition axioms for each operator, and it uses that to help ATP's tasks. Using *do-method*, the following parts of the goal were proven completely automatically.

```
by-induction (forall ?s . inv1 ?s) {
 ...
| (state as (drop1 s)) => (!do (inv1 state) drop1)
| (state as (drop2 s)) => (!do (inv1 state) drop2)
| (state as (dup1 s)) => (!do (inv1 state) dup1)
| (state as (dup2 s)) => (!do (inv1 state) dup2)
| (state as (send2 s)) => (!do (inv1 state) send2)
| (state as (send1 s)) => (!do (inv1 state) send1)
}
```

Some approaches that deal with the automation of algebraic specification methods are briefly presented; A methodology for proving inductive properties of OTSs that aims at automating the proof scores approach to verification, can be found in [97]. In this paper, authors revise the entailment system of proof scores and enrich it with proof rules and tactics. Also, a prototype tool (Constructor-based Inductive Theorem Prover - CITP) implementing the methodology is demonstrated. CITP is implemented in Maude. Another interesting approach is the tool Hets - the Heterogeneous Tool Set [89] that offers parsing, static analysis and proof management. Hets has among others, connections with the algebraic specification languages Maude and CASL and external theorem provers and is based on a graph of logics and logic translations.

In the following we summarize the benefits of the proposed approach and we compare the verification of the properties of the presented case studies, using the proposed methodology and using the proof scores approach.

One advantage of the proposed methodology is that you can use the model-checking tools of Athena to obtain some first insights about the specified system and thus save valuable time. Using the CafeOBJ processor, model checking can be used when the system is expressed in terms of rewriting logic but not directly in equational specifications. Also, the simulate procedure, can become really helpful in understanding how the specified system behaves. Especially when you deal with a complex system where it is almost

impossible to "follow" its execution process, such visualization techniques can provide a clear overview of the system and help in the discovery of possible errors. In addition, the structured proofs supported by Athena can provide valuable information and explanation as to why a property is invariant or not for the specified system. Another important advantage of our approach is that Athena does a thorough check of the overall proof and provides a guarantee that if and when you get a theorem, that result follows logically from the a.b. and the primitive methods (which in this case include the external ATP). This is really helpful because on the other hand, with CafeOBJ's proof scores approach there is much greater room for human oversight. Finally, the automation in the verification is another advantage of the 'Cafe2Athena' methodology. In the case of the Mutex algorithm the desired goal was proven completely automatically. Also, when we used Athena to verify the Alternating Bit Protocol the number of case analyses needed for proving the desired goal was 20 and for proving the additional lemmas was 318. On the other hand, with the Proof Scores approach the number of the required case analyses was 24 and 496, respectively. This makes the total number of case analyses needed in Athena significantly smaller.

Conclusions and Future Directions

In chapter 3, we first presented a Hidden (Sorted) Algebra semantics for production (PR), event condition action (ECA) and knowledge representation rules (KR) as well as for complex event processing, with the goal to have a unifying theory for reactive rules and appropriate tool support for it, built using algebraic specification languages (like CafeOBJ or Maude). Next, we presented a framework for formally specifying reactive rules with the help of the OTS/CafeOBJ method. This framework can express complex systems behaviour while capturing the semantics of the underlying reactive rules, and can be used for the verification of safety properties reactive rule-based agents should meet. In addition, we proposed a methodology, based on rewriting logic specifications written in CafeOBJ, for reasoning about structural errors of systems whose behaviour is expressed in terms of reactive rules and verifying safety properties within the same framework. We then compared the proposed methodologies, based on the different logical systems, through their application in a real-life intelligent agent. Finally, we developed a tool in order to automate the transformation from reactive rules into rewrite transition rules in order to make the verification of reactive rule systems possible for inexperienced users.

Verification support for reactive rule bases can lead to the creation of on line libraries and the definition of operators for combining these rules in a way that preserves the desired properties. Also the formal proofs of the verified properties are executable and could be used as trust credentials for the rule bases agents use, thus allowing the creation of new trust policies and models. Finally, verification techniques could be applied to rule engines and prove the correctness of the implementation of the rules. In the future, we intend to extend the framework in order to be able to model operational reactive systems that need to define an optimized proof-theoretic and operational semantics. Also we plan to integrate the proposed tool and methodologies with other specification systems, like Maude for example, so as to have better tool support and broader use. Finally, we intend to use similar formal techniques in order to verify an open-source rule engine implementation. Thus together with the proposed framework, end-to-end validation will be enabled.

In chapter 4, we argued that due to the increased development of complex context-aware applications in critical domains it is important to be able to formally analyse their behaviour. These systems present new challenges compared to traditional systems and addressing context awareness and self adaptation in a consistent and integrated manner is not a trivial task. For this reason we presented an algebraic framework for their formal specification using Observational Transition Systems (OTSs) specified in the CafeOBJ algebraic specification language.

The proposed framework takes advantage of CafeOBJ's support for para-

metric modules and behavioral object composition method and permits the definition of the core functionality of context-aware adaptive systems as library which can then be used to instantiate a particular system thus reducing the specification effort. It also allows a high degree of modularization which permits the refinement of the components without needing to refine the whole specification. In addition, it represents the relationships between the components of such a system explicitly, which is highly desired for the adoption of the framework in the development of real systems. This methodology supports also the verification of the design of such systems, and can be an effective approach to obtaining verified context-aware software as it permits the re usability not only of the specification code but also of the proofs. This is a critical property because in complex applications, scalability of the methodology becomes a serious concern. As future work, we intend to conduct more case studies using our framework and to expand the created library (e.g. predefine the interaction of different entity systems).

In the last chapter, we proposed an integration of CafeOBJ and Athena environments, with the aim to exploit both the nice properties of CafeOBJ specifications and the soundness and automation in verification offered by Athena, and demonstrated our approach with illustrating case studies. Also we presented several features of the methodology that can be used to: better understand the specified system, model-check desired properties and verify them via theorem proving, both automatically and in a structured and more detailed way. The proposed method aims at combining the strengths of the two languages, by working with OTSs and CafeOBJ but also taking advantage of formal-methods techniques that have traditionally lied outside of the rewriting community.

One future direction is to investigate possible connections with tools incorporating various provers and different specification languages, like Hets for example. Our approach could be used as a vehicle for integrating other algebraic specification methods with more conventional theorem-proving systems based on first- or higher-order logic. Also, when rewriting fails during a proof obligation in a given "proof score" (i.e., when the two sides of a desired identity do not reduce to the same normal form by using all available equations as left-to-right rewrite rules), systems such as CafeOBJ will inform the user of how far the two sides could be rewritten, and this feedback often suggests lemmas that are necessary to complete the proof. As future work, we plan to simulate this process in Athena by translating back from Athena to CafeOBJ in order to help the formulation of intermediate lemmas.

Appendix

- K. Ksystra, N. Triantafyllou, K. Barlas and P. Stefaneas, "An Algebraic Specification of Social Networks", Proceedings SQM 2012. Software Quality Management Conference, Editors: E. Berki, J. Valtanen, P. Nykanen, M. Ross, G. Staples, K. Systa, British Computer Society Quality SG SQM/INSPIRE Conference, Tampere, Finland, pp. 135-146, 2012 (Conference)

- K. Ksystra, N. Triantafyllou and P. Stefaneas, "On the Algebraic Semantics of Reactive Rules". Proceedings RuleML 2012. Antonis Bikakis, Adrian Giurca (Eds.): Rules on the Web: Research and Applications - 6th International Symposium, RuleML 2012, Montpellier, France, August 27-29, 2012, pp. 136-150, Lecture Notes in Computer Science. Springer 2012 (Conference) [Chapter 3]

- K. Ksystra, N. Triantafyllou, P. Stefaneas and P. Frangos: "An Algebraic Framework for Modeling of Reactive Rule-Based Intelligent Agents". Proceedings SOFSEM 2014. Viliam Geffert, Bart Preneel, Branislav Rovan, Julius Stuller, A Min Tjoa (Eds.): Theory and Practice of Computer Science - 40th International Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 26-29, 2014, pp. 407-418, Lecture Notes in Computer Science 8327, Springer 2014 (Conference) [Chapter 3]

- K. Ksystra, N. Triantafyllou and P. Stefaneas, "On Verifying Reactive Rules Using Rewriting Logic". Proceedings RuleML 2014. Antonis Bikakis, Paul Fodor, Dumitru Roman (Eds.): Rules on the Web. From Theory to Applications - 8th International Symposium, RuleML 2014, Co-located with the 21st European Conference on Artificial Intelligence, ECAI 2014, Prague, Czech Republic, August 18-20, 2014, pp 67-81, LNCS Springer 2014 (Conference) [Chapter 3]

- P. Stefaneas, I. Ouranos, N. Triantafyllou and K. Ksystra, "Some Engineering Applications of the OTS/CafeOBJ Method". Proceedings SAS 2014. Specification, Algebra, and Software. A Festschrift Symposium in Honor of Kokichi Futatsugi 2014 Kanazawa, Japan. eds: S. Iida, J. Meseguer, K. Ogata, pp. 541-559, Springer LNCS Festschrift 2014 (Conference)

- N. Triantafyllou, K. Ksystra, P.Stefaneas, A. Kalampakas "Towards Formal Representation and Comparison of Video Content Using Algebraic Semiotics". Proceedings SMAP 2012. 9th International Workshop on Semantic and Social Media Adaptation and Personalization, Corfu, November 6-7, 2014, pp. 48-53, IEEE 2014 (Conference)

- K. Ksystra, P. Stefaneas and P. Frangos, “An Algebraic Approach for the Verification of Context-Aware Adaptive Systems”, International Journal of Software Engineering and Knowledge Engineering, World Scientific, Volume 25, Issue 07, pp. 1105-1128, September 2015 (Journal) [Chapter 4]

- K. Ksystra and P. Stefaneas, “Formal analysis and verification support for reactive rule-based Web agents”. International Journal of Web Information Systems, Volume 12, Issue 4, pp. 418-447, 2016 (Journal) [Chapter 3]

- K. Arkoudas, K. Ksystra, N. Triantafyllou and P. Stefaneas, “Integrating Athena with Algebraic Spcifications” Preliminary Proceedings 22nd WADT, International Workshop on Algebraic Development Techniques, 4-7 September 2014, in memoriam of Joseph Goguen, Sinaia, Romania, Technical Report Simion Stoilow Institute of Mathematics, Romanian Academy, eds R. Diaconescu, M. Codescu, I Tutu. pp 12-13, Sinaia 2014 (Technical Report) [Chapter 5]

- K. Ksystra, N. Triantafyllou and P. Stefaneas, “Combining Algebraic Specifications with First-Order Logic via Athena”, to appear in Algebraic Modeling of Topological and Computational Structures and Applications, Book at Springer Proceedings in Mathematics and Statistics (Book) [Chapter 5]

References

[1] Formal methods, http://users.ece.cmu.edu/koopman/

[2] CafeOBJ Algebraic Specification and Verification, https://cafeobj.org/

[3] Professor Kokichi Futatsugi Personal Page, http://www.jaist.ac.jp/ kokichi/

[4] Berners-Lee, T., Hendler, J., Lassila, O. (2001) The Semantic Web. Scientific American 284, 34-43.

[5] Franklin, S. and Graesser, A., Is it an agent or just a program?: a taxonomy for autonomous agents", in: Proceedings of the third international workshop on agent theories, architectures and languages, Springer-Verlag, 1996.

[6] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, John S. Fitzgerald: Formal methods: Practice and experience. ACM Comput. Surv. 41(4) (2009)

[7] R. Diaconescu and K. Futatsugi, *CafeOBJ report: the language, proof techniques, and methodologies for object-oriented algebraic specification* (AMAST series in computing, Singapore, World Scientific, 1998)

[8] J. A. Goguen and G. Malcolm, A hidden agenda, Technical Report No. CS97-538 (Ed.: University of California at San Diego, 1997).

[9] Razvan Diaconescu and Kokichi Futatsugi. Behavioural coherence in object-oriented algebraic specification. J. Universal Computer Science, 6(1):74–96, 2000. First version appeared as JAIST Technical Report IS-RR-98-0017F, June 1998.

[10] Rolf Hennicker and Michel Bidoit. Observational logic. In A. M. Haeberer, editor, Algebraic Methodology and Software Technology, number 1584 in LNCS, pages 263– 277. Springer, 1999. Proc. AMAST'99.

[11] Goguen, J.A., Diaconescu, R.: Towards an Algebraic Semantics for the Object Paradigm . In: Ehrig, H., Orejas, F. (eds.) 10th Workshop on Abstract Data Types (1994)

[12] R. Diaconescu and K. Futatsugi, Logical foundations of cafeobj, *J. Theoretical Computer Science*, **285**(2) (2002) 289–318.

[13] Iida, S., Futatsugi, K., Diaconescu, R.: Component based algebraic specifications - behavioural specification for component based software engineering. In: In Seventh OOPSLA Workshop on Behavioral Semantics of OO Business and System Specification. (1999)

[14] S. Iida, M. Matsumoto, R. Diaconescu, K. Futatsugi, and D. Lucanu, Concurrent Object Composition in CafeOBJ, Technical Report No. IS-RR-98-0009S (Ed.: Japan Advanced Institute of Science and Technology, 1998)

[15] R. Diaconescu, Behavioural specification for hierarchical object composition, *J. Theoretical Computer Science*, **343**(3) (2005) 305–331.

[16] Diaconescu, R., Futatsugi, K., Iida, S.: Component-based algebraic specification and verification in cafeobj. In: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume II. FM '99, London, UK, UK, Springer-Verlag (1999), 1644-1663

[17] Goguen J., A.: Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations. Theoretical Computer Science, 217–273 (1992)

[18] OMG, Unified modeling language specification version 1.4, 2001.

[19] Futatsugi K., Gaina, D., Ogata, K.: Principles of proof scores in CafeOBJ. Theoretical Computer Science 464, 90–112 (2012)

[20] Ogata, K., Futatsugi, K.: Theorem Proving Based on Proof Scores for Rewrite Theory Specifications of OTSs. Specification, Algebra, and Software, Essays Dedicated to Kokichi Futatsugi, LNCS 8373, 630-656 (2014).

[21] Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science, 96(1):73–155, (1992).

[22] Diaconescu, R., Futatsugi, K., and Iida, S.: CafeOBJ Jewels. In: Futatsugi, K., Nakagawa, A.T., and Tamai, T., editors, CAFE: An Industiral-Strength Algebraic Formal Method, Elsevier, 33–60 (2000)

[23] Ogata, K., Futatsugi, K.: Proof Scores in the OTS/CafeOBJ Method. In: 6th IFIP WG6.1 Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems, pp. 170–184. LNCS 2884, Springer (2003)

[24] K. Ogata and K. Futatsugi, Some tips on writing proof scores in the ots/cafeobj method, *Proc. Conf. on Algebra, Meaning, and Computation* **4060** (2003) 596–615.

[25] Ogata K. and Futatsugi K., Compositionally Writing Proof Scores of Invariants in the OTS/CafeOBJ Method, *Journal of Universal Computer Science*, **19**(6) (2013) 771–804.

[26] K. Futatsugi, J. A. Goguen, and K. Ogata, Verifying Design with Proof Scores, *Proc. Conf. Verified Software: Theories, Tools, Experiments*, Zurich, Switzerland, LNCS, (2005) 277–290.

[27] B. Berstel, P. Bonnard, F. Bry, M. Eckert, and P. Patranjan, Reactive Rules on the Web, Reasoning Web Volume 4636 of the series Lecture Notes in Computer Science pp 183-239 (2007)

[28] Paschke, A., Kozlenkov, A., Boley, H.: A Homogeneous Reaction Rule Language for Complex Event Processing. In: VLDB '07 (2007)

[29] Paschke, A., Boley, H.: Rules Capturing Events and Reactivity. In: Giurca, A., Gasevic, D., Taveter, K. (eds.) Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches., IGI Publishing, May 2009, pp. 215–252 (2009)

[30] Boley, H., Paschke, A., Shafiq, O.: RuleML 1.0: The Overarching Specification of Web Rules. In: RuleML'10 Proceedings of the 2010 international conference on Semantic web rules (2010)

[31] Carlson, J., Lisper, B.: An event detection algebra for reactive systems. In: 4th ACM international conference on Embedded software (2004)

[32] Kowalski, R.A., Sergot, M.J.: A logic-based calculus of events. J. New Generation Computing. 4, 67–95 (1986)

[33] McCarthy, J., Hayes, P.J.: Some Philosophical Problems from the Standpoint of Artificial Intelligence . In: Machine Intelligence 4, Michie, D, Meltzer, B. (eds.) Edinburg University Press, pp 463–502 (1969)

[34] Maude Homepage, http://maude.cs.uiuc.edu/

[35] Ksystra, K., Triantafyllou, N., Stefaneas, P.: On the Algebraic Semantics of Reactive Rules. 6th International Symposium, RuleML 2012, Springer, (2012) 136–150

[36] Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice Hall; 1st edition (1995)

[37] Gilbert, D.: Intelligent Agents: The Right Information at the Right Time. IBM Intelligent Agent White Paper

[38] FIPA (Foundation for Intelligent Physical Agents), www.fipa.org

[39] Badica, C., Braubach, L., Paschke, A.: Rule-based Distributed and Agent Systems. 5th international conference on Rule-based reasoning, programming, and applications, RuleML 2011, Springer, (2011) 3–28

[40] Ericsson, A., Berndtsson, M., Pettersson, P.: Verification of an industrial rule-based manufacturing system using REX 1st International Workshop on Complex Event Processing for Future Internet, iCEP-FIS, (2008)

[41] Paschke, A., Boley, H., Zhao, Z., Teymourian, K., and Athan, T.: Reaction RuleML 1.0: Standardized Semantic Reaction Rules. 6th International Symposium, RuleML 2012, LNCS 7438, Springer, (2012) 100–119

[42] Reaction RuleML, http://ruleml.org/reaction

[43] Paschke, A.: ECA-RuleML: An Approach combining ECA Rules with temporal interval-based KR Event/Action Logics and Transactional Update Logics. ECA-RuleML Proposal for RuleML Reaction Rules Technical Goup, (2005)

[44] Paschke, A., and Boley, H.: Rules Capturing Events and Reactivity. Giurca, A., Gasevic, D., Taveter, K. (eds.) Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches., IGI Publishing, (2009) 215–252

[45] Paschke, A.: ECA-LP/ECA-RuleML: A Homogeneous Event-Condition-Action Logic Programming Language. Int. Conf. on Rules and Rule Markup Languages for the Semantic Web, Athens, Georgia, USA, (2006)

[46] Teymourian, K., and Paschke, A.: Semantic Rule-Based Complex Event Processing. International Symposium, RuleML 2009, LNCS 5858, Springer, (2009) 82–92

[47] Liua, J., Zhangb, J., and Hub, J.: A case study of an inter-enterprise workflow-supported supply chain management system. Information and Management 42, (2005) 441–454

[48] cafeobj@ntua.blogspot.com

[49] Vlahavas, I., Bassiliades, N.: Parallel, object-oriented, and active knowledge base systems. Kluwer Academic Publishers, Norwell, MA, USA, (1998)

[50] Paschke, A., and Kozlenkov, T.: Rule-Based Event Processing and Reaction Rules. International Symposium, RuleML 2009, LNCS 5858, Springer, (2009) 53–66

[51] Xudong, H., Chu, C., Yang, H., and Yang, S. J. H.: A New Approach to Verify Rule-Based Systems Using Petri Nets. Information and Software Technology, vol. 45, issue 10, (2003) 663–669

[52] Patkos, T., Chrysakis, I., Bikakis, A., Plexousakis, D., Antoniou, G.: A Reasoning Framework for Ambient Intelligence. S. Konstantopoulos et al. (Eds.): SETN 2010, LNAI 6040, Springer-Verlag, (2010) 213–222

[53] Ogata K., Futatsugi, K.: Proof Score Approach to Analysis of Electronic Commerce Protocols, *Int. J. Soft. Eng. Knowl. Eng.*, **20**(253) (2010) 253–287.

[54] Ogata, K., Futatsugi, K.: Proof score approach to verification of liveness properties. IEICE Transactions E91-D, 2804–2817 (2008).

[55] Ksystra, K., Stefaneas, P., Frangos, P.: An Algebraic Framework for Modeling of Reactive Rule-Based Intelligent Agents. SOFSEM: Theory and Practice of Computer Science - 40th International Conference on Current Trends in Theory and Practice of Computer Science, LNCS, 407-418 (2014).

[56] Jin X., Lembachar Y., Ciardo G.: Symbolic verification of ECA rules. International Workshop on Petri Nets and Software Engineering (PNSE'13) and International Workshop on Modeling and Business Environments (ModBE'13), 41-59 (2013)

[57] Fors, T.: Visualization of rule behaviour in active databases. VDB, 215–231 (1995).

[58] Benazet, E., Guehl, H., Bouzeghoub, M.: Vital: A visual tool for analysis of rules behaviour in active databases. Second International Workshop on Rules in Database Systems, Springer-Verlag, 182–196 (1995).

[59] Berstel, B., Leconte, M. : Using Constraints to Verify Properties of Rule Programs. ICST Third International Conference on Software Testing, Verification and Validation, Paris, France (2010).

[60] K. Ksystra and P. Stefaneas, Formal analysis and verification support for reactive rule-based Web agents. International Journal of Web Information Systems, Volume 12, Issue 4, pp. 418-447, 2016

[61] K. Ksystra, N. Triantafyllou and P. Stefaneas, "On Verifying Reactive Rules Using Rewriting Logic". Proceedings RuleML 2014. Antonis Bikakis, Paul Fodor, Dumitru Roman (Eds.): Rules on the Web. From Theory to Applications - 8th International Symposium, RuleML 2014, Co-located with the 21st European Conference on Artificial Intelligence, ECAI 2014, Prague, Czech Republic, August 18-20, 2014, pp 67-81, LNCS Springer 2014

[62] M. Janik, A. Nanda, R. Rabbani, Semantic Context Specification.

[63] [https://www.interaction-design.org/literature/book/the-encyclopedia-of-human-computer-interaction-2nd-ed/context-aware-computing-context-awareness-context-aware-user-interfaces-and-implicit-interaction]

[64] N. Triantafyllou, Software Engineering Applications of the OTS/-CafeOBJ Algebraic Specification Method, Phd Thesis (National Technical University of Athens, 2014).

[65] I. Ouranos, K. Ogata, and P. Stefaneas, TESLA source authentication protocol verification experiment in the Timed OTS/CafeOBJ method: Experiences and Lessons Learned, *IEICE Transactions*, Tokyo, (2014), Accepted for publication.

[66] P. Stefaneas, I. Ouranos, N. Triantafyllou, and K. Ksystra, Some Engineering Applications of the OTS/CafeOBJ Method, *Proc. Conf. Specification, Algebra, and Software A Festschrift Symposium in Honor of Kokichi Futatsugi*, Japan, LNCS, Springer, (2014), Accepted for publication.

[67] M. U. Iftikhar and D. Weyns, A Case Study on Formal Verification of Self-Adaptive Behaviors in a Decentralized System, in *Proc. Conf. on Foundations of Coordination Languages and Self-Adaptation*, UK, (2012) 45–62.

[68] D. Weyns, S. Malek, and J. Andersson, FORMS: a formal reference model for self-adaptation, in *Proc. Conf. on Autonomic computing*, USA (2010) 205–214.

[69] A. Coronato and G. De Pietro, Formal Specification and Verification of Ubiquitous and Pervasive Systems, *J. ACM Transactions on Autonomous and Adaptive Systems* **6**(1) (2011) 1–9.

[70] N. Khakpour and S. Jalili, PobSAM: Policy-based Managing of Actors in Self-Adaptive Systems, in *6th Int. Workshop on Formal Aspects of Component Software*, Eindhoven, (2009) 129–143.

[71] K. Schneider, T. Schuele, and M. Trapp, Verifying the adaptation behavior of embedded systems, in *Proc. Conf. on Self-adaptation and self-managing systems*, China (2006) 16–22.

[72] M. Hussein, J. Han, and A. Colman A, Specifying and Verifying the Context-aware Adaptive Behavior of Software Systems, Technical Report No. C3-516-03 (Ed.: Australia: Swinburne University of Technology, 2010).

[73] R. Adler, I. Schaefer, T. Schuele, and E. Vecchié, From Model-Based Design to Formal Verification of Adaptive Embedded Systems, in *Proc. International Conf. on Formal Engineering Methods*, USA (2007) 76–95.

[74] A. Rakib and R. U. Faruqui, A Formal Approach to Modelling and Verifying Resource-Bounded Context-Aware Agents, in *Proc. Conf. on Context-Aware Systems and Applications*, Vietnam (2012) 86–96.

[75] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C., Composing Adaptive Software, *J. Computer*, **37**(7) (2004), 56–64.

[76] T. Nipkow, L. C. Paulson, and M. Wenzel, Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS, Springer **2283** (2002).

[77] Y. Bertot and P. Casteran, Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions, Springer, (2004).

[78] Seino, T., Ogata, K., Futatsugi, K.: A toolkit for generating and displaying proof scores in the OTS/CafeOBJ method, *ENTCS*, **147**(1) (2006) 57–72.

[79] Nakano, M., Ogata, K., Nakamura, M., Futatsugi, K.: Creme: an Automatic Invariant Prover of Behavioral Specifications, *Int. J. Soft. Eng. Knowl. Eng.*, **17**(6) (2007) 783–804.

[80] K. Ogata, M. Nakano, M. Nakamura, and K. Futatsugi, Chocolat/SMV: A translator from CafeOBJ into SMV, in *Proc. Int. Conf. on Parallel and Distributed Computing Applications and Technologies*, China (2005) 416–420.

[81] W. Kong, K. Ogata, T. Seino, and K. Futatsugi, A lightweight integration of theorem proving and model checking for system verification, in *Proc. 19th Asia-Pacific Conf. on Software Engineering*, Taiwan (2005) 59–66.

[82] Arkoudas, K.: Athena, proofcentral.org, (2004)

[83] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada.: Maude: Specification and Programming in Rewriting Logic. Maude System documentation. March, 1999.

[84] CASL, Web page, http://www.casl.umd.edu/about

[85] HOL/Isabelle, Web page, http://isabelle.in.tum.de/library/HOL/

[86] T. Nipkow: Programming and Proving in Isabelle/HOL. Technical Report, 2014.

[87] S. Autexier, T. Mossakowski.: Integrating HOL-CASL into the Development Graph Manager MAYA. Frontiers of Combining Systems. Lecture Notes in Computer Science Volume 2309, 2002, 2–17.

[88] Hets, Web page, http://theo.cs.uni-magdeburg.de/Research/Hets.html

[89] M. Codescu, F. Horozal, M. Kohlhase, T.l Mossakowski, F. Rabe, K. Sojakova.: Towards Logical Frameworks in the Heterogeneous Tool Set Hets. In Till Mossakowski, Hans-Jörg Kreowski (Eds.), Recent Trends in Algebraic Development Techniques, 20th International Workshop, WADT 2010, Vol. 7137, pp. 139–159, Lecture Notes in Computer Science. Springer.

[90] Musser, D.: Understanding Athena Proofs.

[91] Vampire, Web page, www.vprover.org/

[92] Spass, Web page www.spass-prover.org/

[93] F. J. Pelletire. A Brief History of Natural Deduction. History and Philosophy of Logic, 20:1-31, 1999.

[94] T. Mossakowski and A. Tarlecki, A Relatively Complete Calculus for Structured Heterogeneous Specifications, In: Foundations of Software Science and Computation Structures, FOSSACS 2014, LNCS 8412.

[95] M. Aiguier, F. Barbier,: An institution-independent Proof of the Beth Definability Theorem.

[96] Algebraic modeling of topological and computational structures (AlModTopCom), http://www.math.ntua.gr/ sofia/ThalisSite/publications.html

[97] Gaina, D., Lucano, D., Ogata, K., Futatsugi, K.: On Automation of OTS/CafeOBJ Method. In: Specification, Algebra, and Software, LNCS 8373, 578-602 (2014)

[98] N. Preining, Algebraic specifications and Functional programming with CafeOBJ, Technical Report, JAIST 2015.

[99] I. Cafezeiro, J. Viterbo, A. Rademaker, E. H. Haeusler, and M. Endler, A Formal Framework for Modeling Context-Aware Behavior in Ubiquitous Computing.

[100] Ogata K., Futatsugi K.: Modeling and verification of real-time systems based on equations. Science of Computer Programming, 66, pp. 162–180 (2007)

[101] Diaconescu, R., Goguen, J., Stefaneas, P.: Logical support for modularization. In: Second annual workshop on Logical Environments (1993)

[102] Goguen, J., Winkler, T., Meseguer, J., Futatsugi, K., Jouannaud, J.P.: Introducing OBJ. Software Engineering with OBJ: Algebraic Specification in Action. Kluwer (2000).