Specification of an Access Control System with a Formalism Combining CCS and CASL

Gwen Salaün, Michel Allemand, and Christian Attiogbé
IRIN, Université de Nantes
2 rue de la Houssinière, B.P. 92208, 44322 Nantes cedex 3, France
{salaun,allemand,attiogbe}@irin.univ-nantes.fr

Abstract

Multi-formalism specifications are essential for the modelling of complex systems including different aspects such as data or concurrency. We advocate a formalism which combines the CCS process algebra with the CASL algebraic specification language. Formal foundations of this combination are presented following two steps, the syntax and the semantics. Our proposal is illustrated with a real size case study: an access control system to a set of buildings. With this concrete example, we aim at showing how our formalism proposal could be used to specify a comprehensive application.

Keywords: Multi-formalism Specifications, CCS, CASL, Formal Foundations.

1 Introduction

All the aspects of complex software systems can neither be specified nor verified with only one approach. The joint use of several different formal methods, called *multi-formalism specifications*, is necessary for the design of such systems. This promising orientation is of major interest for the software engineering community.

The main motivation is that the different parts of software systems have to be specified with appropriate formalisms and have to be formally verified with suitable tools. The use of a single formalism is not sufficient enough to specify the different aspects of complex systems. We need to have at one's disposal adequate formal languages to specify the two main parts of systems that are data (or static) and behavioural (or dynamic) parts. Furthermore, these different parts have to be linked in a formal way.

In this work, we deal with the approach combining a process algebra with an algebraic specification language. Works with similar goals already exist; the well-known results are LOTOS [3] and its successor E-LOTOS [13]. In [1], authors propose a classification of different techniques based on algebraic specifications to specify concurrent systems. Our work direction is the second kind of approach (A2) proposed in this taxonomy, corresponding to approaches integrating a formalism for the concurrent aspects with algebraic specifications of the static data types.

Transversely, for some years, the wish to unify the numerous algebraic specification formalisms has emerged. This has led to the CASL (Common Algebraic Specification Language) language [7]. This expressive language is very promising for the specification of complex data-oriented systems. However, CASL is not suitable to the specification of the dynamic aspects (parallelism, communication, concurrency). The extension of CASL to specify *reactive systems* led, among others, to the Reggio and Repetto proposal [20] which combined CASL and Statecharts. On a wider scale, Astesiano *et al.* [2] present methodological guidelines (which we partially follow) in the design phase of formalisms associating a data description language, particularly CASL, with a paradigm-specific one.

To cope with the CASL limitations with reference to the specification of dynamic aspects, we choose to combine CASL with a process algebra: CCS [16]. From the language point of view, CCS is simple and provides few constructions. However, it is sufficiently expressive to describe dynamic aspects of systems. The central idea of our combination is the extension of Milner's value passing CCS. Algebraic terms written in CASL are used as value passing in agents and actions. Our combination is syntactic in the sense of Fischer's classification [9]. Then, we define a global semantics for the proposed formalism. This approach yields an expressive specification language.

Now, we briefly compare our combination with the previous mentioned languages to reinforce the interests of this proposal. About LOTOS, the ACT ONE part has been often criticized [11] (for example the lack of proper structuring mechanisms). The use of CASL improves the expressiveness of the data part. Moreover, CCS has a minimal set, but sufficiently expressive, of simple operators to specify concurrent aspects of systems. E-LOTOS, an enhanced proposal for LOTOS, takes into account several improve-

0-7695-1573-8/02/$17.00 (C) 2002 IEEE

ments about the data part (partial operations for instance), the dynamic part (priorities, exception raising, etc), and the whole organization (modules). However, a final version is not widely admitted, and is always in balloting. In addition, from our point of view, E-LOTOS suffers from some technical shortcomings (for example the dynamic typing), and above all is too complex. Concerning the Casl-Chart combination, a great drawback of this language is the numerous semantics existing for the Statecharts, and the lack of formality in some of them. Other approaches aim at combining process algebras with state oriented languages such as Z or B. An example of work focusing on this topic is the proposal of Treharne and Schneider [25].

The remainder of this paper is organized as follows. In Section 2, we briefly describe the formal foundations of our combination: syntax and semantics. Then, in Section 3, we illustrate the practical use of this formalism on a case study about an access control system. An extended version of this paper is available in [23].

2 Overview of the Formal Foundations

In this section, we introduce the formal basis underlying the definition of the formalism combining CCS and CASL. We focus respectively on the syntactic aspects and on the operational semantics giving the meaning to the whole language.

2.1 Formal Syntax

CCS [16] allows synchronous, binary, and oriented communication. There are operators for prefixing (.), non deterministic choice ($+$), parallel composition ($|$), restriction to enforce synchronization ($\backslash$), *if* structure, relabeling (E[f]), and extended sum ($\Sigma_{i \epsilon I}$ E_i). We emphasize that our approach is control driven, and CCS is called the main formalism because it gives the behaviour of the full specification.

Concerning CASL, the whole language is considered, *i.e.* basic, structured, architectural specifications, and libraries. Yet, some restrictions are necessary at the semantic level to perform term rewriting (see subsection 2.2). The syntax of our language is formalized in Figure 1 using a BNF-like notation. The + and | symbols of the BNF-like notation must not be confused with the ones of the CCS notation. Therefore, the non deterministic choice and the parallel composition of CCS are written larger than the others.

We finish with some precisions about interactions between CCS agents and data expressed with CASL. The value passing is described by CASL expressions which are inserted in the CCS syntax at five levels: the parameterized agent declarations and calls, the input and output parameterized actions, and the condition of the *if* operator.

SPEC	::=	CASL-SPEC CCS-SPEC
CASL-SPEC	::=	*appendix A of CASL summary [7]*
CCS-SPEC	::=	AGENT+
AGENT	::=	AGENT-ID $\stackrel{def}{=}$ BEHAVIOUR | AGENT-ID(VAR-DECL+) $\stackrel{def}{=}$ BEHAVIOUR
BEHAVIOUR	::=	0 | PREFIXING | BEHAVIOUR$+$BEHAVIOUR | BEHAVIOUR$\Big|$BEHAVIOUR | BEHAVIOUR$\backslash${ACTION+} | AGENT-CALL | (BEHAVIOUR) | if CASL-PRED then BEHAVIOUR | BEHAVIOUR [f] | $\Sigma_{i \epsilon I}$ BEHAVIOUR$_i$
PREFIXING	::=	ACTION.BEHAVIOUR | $\overline{\text{ACTION}}$.BEHAVIOUR | ACTION(VAR-DECL+).BEHAVIOUR | $\overline{\text{ACTION}}$(APPLICATION+).BEHAVIOUR | τ.BEHAVIOUR
AGENT-CALL	::=	AGENT-ID | AGENT-ID(APPLICATION+)
APPLICATION	::=	OP-NAME EXPR* | VAR
CASL-PRED	::=	PRED-NAME EXPR* | VAR
EXPR	::=	VAR | APPLICATION

Figure 1. Syntax of the combination

2.2 Formal Semantics

In this subsection, we introduce the global operational semantics for our formalism. We follow an approach similar to Galloway's one for ZCCS [10]. The precise meaning of the different CCS constructions with CASL value passing is given using inference rules. The semantics of the algebraic expressions in the CCS specification is defined using terms rewriting.

The environment E is a context memorizing different informations used to define the inference rules. E is a couple composed of the rewrite rules R deduced from the axioms of the CASL algebraic part, and of a set of tuples $CCSE$ for CCS agents. This set is built from the full specification (*i.e.* both CASL sorts and CCS agents).

$$\boxed{E \triangleq < R, CCSE >}$$

The semantics of the independent CASL specifications is as described in [8], but the meaning of CASL terms appearing in the CCS agents is given by rewriting of closed terms. This choice is justified since it is suitable to an operational semantics. The rewriting is performed using a set R of rewrite rules deduced from the CASL specifications. This can be achieved following the conceptual steps defined in [14], which is rather devoted to tools; nevertheless this shows how rewriting of CASL terms can be done.

0-7695-1573-8/02/$17.00 (C) 2002 IEEE

In [14], Ringeissen and Kirchner wish to execute CASL equational specifications with the ELAN rewrite engine [4]. Both basic and structured specifications are considered, even though some restrictions are assumed (subsorting and partiality features). The translation from CASL to ELAN is performed using intermediate formats (FCasEnv and Efix ATerms). Then, a last step of translation transforms Efix ATerms into the ELAN executable format (REF programs).

The set $CCSE$ contains informations memorized during the agents declaration. More precisely, for each agent declaration, we store in this set a tuple containing the agent name (or agent constant) AC, its whole behaviour H, and the list of the agent parameters (identifiers) AP.

$$CCSE \triangleq \{< AC_1, H_1, AP_1 >, \ldots, < AC_n, H_n, AP_n >\}$$

These three values, associated with each agent, are useful in presence of agents calls. In such a case, the agent constant is substituted by the behaviour corresponding to this call. Moreover, for a parameterized agent, the identifiers memorized during the declaration are substituted in the whole behaviour by the terms in parameter.

There are two kinds of inference rules in our operational semantics. The first one corresponds to the construction of the E environment from the agents definitions and the algebraic specification part. The E environment, after being completely built, is never modified. The second one gives the meaning for each (possibly extended) CCS operator. For this second type, the global specification is seen as a Labelled Transitions System (LTS) which evolves by the application of the inference rules. For instance, the next rules give successively the meaning of the parallel composition, the conditional operator, and the agent call.

```
BEHAVIOUR ::= BEHAVIOUR|BEHAVIOUR
```

$$\frac{F \xrightarrow{\alpha} F'}{F|G \xrightarrow{\alpha} F'|G}$$

If a behaviour F evolves by the action α into F', then the parallel composition $F|G$ evolves by α into $F'|G$. The dual rule for the symmetrical case is omitted.

$$\frac{F \xrightarrow{a} F' \quad G \xrightarrow{\overline{a}} G'}{F|G \xrightarrow{\tau} F'|G'}$$

If a behaviour F evolves into F' after the firing of the input action a, and a behaviour G evolves into G' after executing the output action $\overline{a}$, then $F|G$ evolves into $F'|G'$ by τ. The dual rule, in which F does an output action and G an input action, is omitted, as in the next rule.

$$\frac{F \xrightarrow{a(x_1:S_1,\ldots,x_n:S_n)} F' \quad G \xrightarrow{\overline{a}(t_1,\ldots,t_n)} G'}{F|G \xrightarrow{\tau} F'[\overset{R}{\downarrow} t_1/x_1, \ldots, \overset{R}{\downarrow} t_n/x_n]|G'}$$

If a behaviour F evolves into F' after the firing of an input parameterized action, and a behaviour G evolves into G' by an output parameterized action, then $F|G$ evolves into $F'|G'$ by τ. The input variables x_i are substituted in the behaviour F' by the terms t_i received during the communication. These terms are rewritten in their normal forms thanks to the R rewrite rules (part of the E environment).

```
BEHAVIOUR ::=  if CASL-PRED then BEHAVIOUR
```

$$\frac{F \xrightarrow{\alpha} F' \quad \overset{R}{\downarrow} p = true}{\texttt{if } p \texttt{ then } F \xrightarrow{\alpha} F'}$$

If F becomes F' by the firing of α, and the condition p (predicate with closed terms), rewritten with R, is evaluated to $true$, then the conditional expression with the behaviour F joined to the block `then` becomes F' after the firing of α. If the condition is evaluated to $false$, the behaviour becomes 0, *i.e.* the agent which cannot do anything (termination). Note that the p rewriting does not need a variables binding environment because, when it is performed, there are no free variables in the predicate due to preliminary variables substitutions.

```
AGENT-CALL ::=  AGENT-ID
```

$$\frac{F \xrightarrow{\alpha} F' \quad < AC, F, AP > \in CCSE}{AC \xrightarrow{\alpha} F'}$$

If a behaviour F evolves by α into F', and the tuple $< AC, F, AP >$ is in the $CCSE$ set, then the unparameterized agent AC evolves by α into F'.

```
AGENT-CALL ::=  AGENT-ID(APPLICATION+)
```

$$\frac{F[\overset{R}{\downarrow} t_1/AP[1], \ldots, \overset{R}{\downarrow} t_n/AP[n]] \xrightarrow{\alpha} F' \quad < AC, F, AP > \in CCSE}{AC(t_1, \ldots, t_n) \xrightarrow{\alpha} F'}$$

If a behaviour F, in which the parameters $AP[i]$ are substituted by the normal forms (obtained after rewriting with R) of the real terms t_i, becomes F' by α, and the tuple $< AC, F, AP >$ is in the set $CCSE$, then the parameterized agent AC becomes F' by α.

0-7695-1573-8/02/$17.00 (C) 2002 IEEE

3 Case Study: Access Control Specification

In the previous section, we introduced the formal aspects underlying our proposal of combination. Now, we focus on a practical case, and we show how our formalism can deal with the specification of an access control system. The study is made of two separate parts. First of all, we introduce the necessary CASL data types; then we describe the CCS processes, and particularly an end-example of concrete system. We start with a short presentation of the case study.

3.1 Informal Requirements

This case study is about a computer system to manage entrances and exits of people (for instance, students, teachers, and so on) in a set of buildings, as the buildings of an university campus. The informal requirements document can be found in [15].

Each user is recorded in a group, and each building is open to some groups. Each person who could access to a building has a magnetic card. The set of cards is managed by the system. Every card is either in circulation, or robed, or lost. Each access is bound to a card reader, a door, and a sensor detecting the opening and the closure of the door, or detecting the passage of a person. The card reader has two lights. The first is green and indicates an authorized access. The second is red and corresponds to a rejected access. The door is either blocked or unblocked. Moreover, each access is in a single direction, *i.e.* it allows only either entrances or exits.

The access protocol is the following. The user passes his/her card in the card reader if the two lights are switched off. If the access is allowed, the door is unblocked, the green light is switched on, and the person has thirty seconds for passing through the door. If the user passage is detected, the log is updated (see below for the log), the door is blocked, and the green light is switched off. If the user is not allowed to pass the door, the red light is switched on for two seconds.

In this case study, it is assumed that users respect the protocol (disciplined users). For example, users do not pass their cards in the card reader if this one has a light switched on. For various reasons (frauds detection, fire, incidents, etc), it is required to memorize entrances and exits in/from buildings, in order to know users in a building at a precise moment (present or past). Before detailing the whole specification, we notice that *access* has two different meanings. It can refer to an entrance or an exit, or it can represent the physical place where users go in and out of buildings.

3.2 CASL Data Types

The needed data types for the case study specification are described. We distinguish several basic sorts, and other more complex built from the basic ones. The data types dependence graph is reported in Figure 2. Some examples extracted from the full specification are given to illustrate the informal explanations.

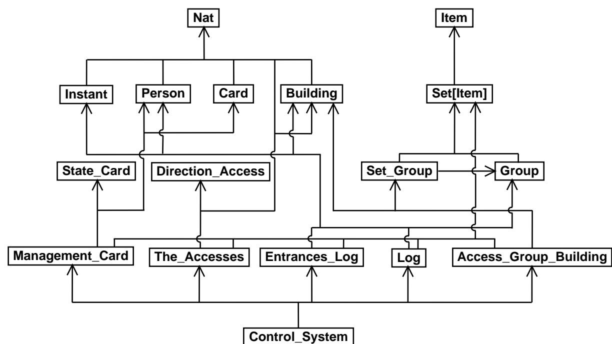

Figure 2. Data types dependence graph

The sort `Nat` represents natural numbers. The sort `Item` is useful to define generic sets: `Set[Item]`. The sort `Instant` indicates entrance and exit moments, delays, and the time passing. It is based on the sort `Nat`.

```
spec INSTANT = NAT with
  sort Nat |-> Instant,
  ops
    0 |-> initial_instant,
    s __ |-> inc __,
    __ - __ |-> dec __ __
 ...
end
```

The sorts `Person` and `Card` identify respectively users and access cards. A card has three possible states (`State_Card`): in circulation, robed or lost. The sort `Direction_Access` fixes the direction of the access. The sort `Building` identifies precisely buildings. Groups of people (`Group`) are modelled by instantiation of generic sets with the sort `Person`. The sort `Set_Group` is declared in the same way as the sort `Group` detailed below.

```
spec GROUP =
  SET[ PERSON
    fit
      sort Item |-> Person,
      pred eq_item __ __ |-> eq_person __ __ ]
  with
    sort Set[Person] |-> Group,
    ops
      empty_set |-> empty_group,
      insert_item __ __ |-> insert_person __ __,
    preds
      eq_set __ __ |-> eq_group __ __,
      is_set_empty __ |-> is_group_empty __,
      is_in_set __ __ |-> is_in_group __ __
end
```

Now, more complex data types are defined. They are obtained by instantiation of generic sets with suitable tuples. The sort `Access_Group_Building` binds to each building groups of people allowed to access it. To build this sort, a data type is locally defined in the specification. It corresponds to a record made of a building and a set of groups. This local definition is justified because it is very specific to the current system, and its reuse is so restricted. Then,

0-7695-1573-8/02/$17.00 (C) 2002 IEEE

the sort Set is instantiated with entities of this type. This construction approach is also chosen for the following sorts.

The sort Entrances_Log contains people recorded in a building at the present instant. The sort Log corresponds to a set of tuples constituted of a person, a building, an instant of entrance, and an instant of exit. Management_Card binds to each card its owner, and its state.

The sort The_Accesses describes the set of accesses managed by the system. Each access is made of an identifier (a natural number), a building, and a direction. An operation provides the building for a given access identifier. Two predicates indicate if an access is an entrance or an exit.

```
spec THE_ACCESSES =
  SET [ NAT and BUILDING and DIRECTION_ACCESS then
    generated type Tuple_IBDA ::= __ __ __
      ( the_id __ :  Nat;
      the_building __ :  Building;
      the_direction __ :  Direction_Access )
 ...
  with
    sort Set[Tuple_IBDA] |-> The_Accesses,
    ops
      empty_set |-> empty_access_set,
      add_item __ __ |-> add_access __ __,
      insert_item __ __ |-> insert_access __ __
  then
  op
    building_of_id __ __ :
        Nat * The_Accesses ->?  Building;
  preds
    is_entrance_access __ __ :  Nat * The_Accesses;
    is_exit_access __ __ :  Nat * The_Accesses;
 ...
end
```

The last sort Control_System gathers the previous five sets in a single structure. This one could perform tests to deliver buildings access authorizations. Two predicates, one for entrances and another for exits, achieve these verifications for a given card and a given access identifier. Several operations are useful to update the logs for each real passage. For instance, remove_an_entrance_cs removes a record from the entrances log for a given person. This operation is useful when a user leaves a building.

```
spec CONTROL_SYSTEM =
  ACCESS_GROUP_BUILDING and MANAGEMENT_CARD and
  ENTRANCES_LOG and LOG and THE_ACCESSES then
  generated type Control_System ::= system __ __ __ __ __
    ( the_allowed_accesses __ :  Access_Group_Building;
    the_management_of_cards __ :  Management_Card;
    the_log_of_entrances __ :  Entrances_Log;
    the_log __ :  Log;
    the_accesses __ :  The_Accesses )
  preds
    authorization_of_entrance __ __ __ :
        Card * Nat * Control_System;
    authorization_of_exit __ __ __ :
        Card * Nat * Control_System;
  ops
    remove_an_entrance_cs __ __ :
        Person * Control_System -> Control_System;
 ...
```

All the behavioural aspects can now be specified thanks to the CCS formalism with value passing expressed by CASL terms. We detail this second part of the specification in the next subsection.

3.3 CCS Agents

We present successively all the necessary processes for the modelling of the dynamic part of the access control system. The handling of CASL terms by CCS agents is of major interest.

A door is either blocked or locked.

$$\mathrm{Door}_i \stackrel{def}{=} \mathrm{ublkDoor}_i.\mathrm{blkDoor}_i.\mathrm{Door}_i + \mathrm{lockDoor}_i.\mathrm{releaseDoor}_i.\mathrm{Door}_i$$

A sensor of passage indicates to its controller if a user passes the access.

$$\mathrm{DoorSensor}_i \stackrel{def}{=} \overline{\mathbf{passDoor}_i}.\mathrm{DoorSensor}_i$$

A card reader synchronizes itself with a user when (s)he passes his/her card. Either the card is identified, or it is not recognized. In the first case, the card reader synchronizes with the user, and after with its controller. The card identifier is sent to the controller so that it tests if the card owner is allowed to pass the access. After this test, the card reader switches on the green light for an authorization, and the red one otherwise. If the card is not identified, the red light is switched on. Similarly, if the controller decides to lock the access, the red light is switched on until the release.

$$\mathrm{CardReader}_i \stackrel{def}{=} \mathrm{passCardValid}_i(\mathrm{c{:}Card}).\overline{\mathbf{alertCtlrCardValid}_i(\mathbf{c})}.(\mathrm{switchOnGnLt}_i.\mathrm{switchOffGnLt}_i.\mathrm{CardReader}_i + \mathrm{switchOnRdLt}_i.\mathrm{switchOffRdLt}_i.\mathrm{CardReader}_i) \ldots$$

The agent Timer has two possible behaviours. Firstly, it can receive a message from the controller indicating a waiting time to be simulated. Consequently, the auxiliary process TimerA performs the delay. Secondly, it can communicate with the clock. Then, the clock synchronizes with every timer, at each incrementation of the clock, in order to decrease the waiting time of the timers which simulate a delay (see the explanations of the agent Clock in the following).

$$\mathrm{Timer}_i \stackrel{def}{=} \mathrm{wait}_i(\mathrm{t{:}Instant}).\mathrm{TimerA}_i(\mathrm{t}) + \mathrm{incClock}.\mathrm{Timer}_i$$

The process TimerA has a parameter which corresponds to the remaining waiting time. It provides three possible behaviours. Firstly, when the time is not entirely elapsed, it calls itself recursively, and decrement the number of remaining time units. Secondly, the delay is finished, and the controller is informed. Thirdly, an event to stop the delay is received from the controller. This last possibility occurs only in the case of thirty seconds waiting: the interruption is possible only if the user passes the door before the timeout of thirty seconds. In the two last cases, the process Timer

0-7695-1573-8/02/$17.00 (C) 2002 IEEE

is called in order to offer the possibility to perform a new delay asked by the controller. In the following, we choose to write CASL terms with the infix notation.

```
TimerA_i(t:Instant) =
  if (eq_instant t initial_instant)
     then incClock.TimerA_i(dec t 1)
  + if (t <= initial_instant) then isFinished_i.Timer_i
  + stopWaiting_i.Timer_i
```

The agent `User` has as parameter the card to pass in the card reader. This parameter is bound to the action every time the card is correctly passed and well recognized by a card reader. This agent is not useful to specify the system, but it introduces actions on which users and the system interact.

```
User(c:Card) =def
  passCardValid_i(c).User(c)
  + passCardNoValid_i.User(c)
```

The clock has two possible behaviours. First of all, it can increment the current time. In this case, it synchronizes with every timer, to decrement them, via an auxiliary agent `ClockAux`. The other behaviour of this process is to communicate the current moment to the tester so that the tester completes the logs after a passage. Indeed, it is the agent `Tester` which possesses the CASL sort `Control_System` described previously. The agent `Clock` has as parameter the current time represented by an instant, the number of timers of the system (*i.e.* the access number), and the number of timers already warned of the current time incrementation.

```
Clock(t:Instant, nbt:Nat, nbat:Nat) =def
  incClock.ClockAux(inc t, nbt, nbat-1)
  + giveHour(t).Clock(t, nbt, nbat)
```

The auxiliary agent `ClockAux` proposed as many synchronizations as there are timers. When each timer has been informed, the agent `Clock` is called. At any time, this agent can synchronize itself with the tester if this one needs the current moment to complete one of the logs.

The repetition of the synchronization on the action `incCLock` is due to the lack of multi-way synchronization in CCS (only binary one). This is a choice in the CCS design. Furthermore, we assume that all these synchronizations execute themselves between two units of logical time to preserve the specification consistency. These hypotheses are non formal, then that could lead to inconsistencies. This drawback is due to the lack of real-time constraints possibilities. A solution would be using a version of timed CCS [17] in order to express the possible temporal aspects in a formal way. This is one of the main direction for future works.

```
ClockAux(t:Instant, nbt:Nat, nbat:Nat) =def
    ( if (nbt <= 1) then incClock.Clock(t, nbt, nbt)
    + if not (nbt <= 1)
        then incClock.ClockAux(t, nbt, nbat-1) )
    + giveHour(t).Clock(t, nbt, nbat)
```

The tester has as parameter the data type `Control_System`. It contains all the informations to perform the passage authorization tests. This agent receives the requests of the controller by the action `test` to which is joined the user card and the access identifier. The CASL predicates are used to manage the tests. After, the tester executes an action representing a positive or negative answer. In case of accepted accesses, logs are updated. Four cases are possible according to the access direction and the test result. We detail only one possible case below: entrance access with positive answer. In this situation, the tester gets the current moment thanks to the clock, and completes the entrances log with this new access.

```
Tester(syst:Control_System) =def
  test(c:Card, i:Nat).
  (
    if ( is_entrance_access i (the_accesses syst) /\
      authorization_of_entrance c i syst )
    then canPass_i.giveHour(t:Instant).
      Tester(add_entrance_cs (person_having_card c
         (the_management_of_cards syst))
         (building_of_id i (the_accesses syst)) t syst)
    + ...
```

The controller receives an event from the card reader which states the passage of a card. If the card is valid (well recognized), the controller asks to the `Tester` agent if the owner of this card is allowed to pass.

If the user can pass the access, the door is unblocked, and the green light is switched on. A waiting of thirty seconds starts. It is stopped either by the passage detection of a person by the sensor, or because the thirty seconds are elapsed. In the two cases, the green light is switched off and the door is blocked.

If the tester indicates that the user is not allowed to pass the door, the red light is switched on during two seconds. Similarly, if the passage in the card reader of a card is not valid, the red light is switched on during two seconds. The controller can also lock the door. In this situation, the red light is switched on. The access will be available only after the release.

```
Controller_i =def
  alertCtlrCardValid_i(c:Card).test(c,i).
  (
    canPass_i.ublkDoor_i.switchOnGnLt_i.wait_i(thirtySec).
   ...
```

We remind on Figure 3 the complementary actions on which agents synchronize themselves. The aim of this figure is just to indicate these interactions, and not to show another informations (*e.g.*: call of an agent by another, parameters of processes). Boxes represent agents, and the links between boxes correspond to communicating actions. Some writing simplifications are achieved to clarify the diagram. For instance, all the communications between the controller and the card reader are not detailed. Furthermore, the clock and the auxiliary clock are gathered in a single box because these agents communicate with the same actions.

0-7695-1573-8/02/$17.00 (C) 2002 IEEE

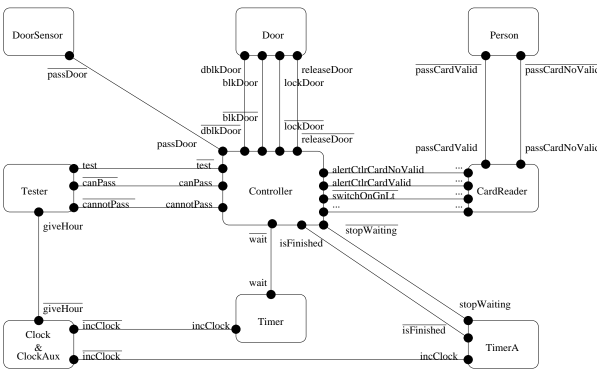

Figure 3. Synchronizations between agents

From the previous declaration of agents, real systems could be specified, depending on the number of buildings, accesses (entrances and exits), and users. Parallel composition and restriction operators are sufficient to represent possible complete system. Parameters are written with CASL terms. Internal communications (not observable by the outer environment) are described by inserting the concerned actions in the restriction set.

```
System ≝
  (Door₁|Door₂|DoorSensor₁| ... |User(card (s 0))|
   Timer₁ | Timer₂ | Clock (initial_instant) |
   Tester (system (add_access (building (id (s 0) ...
    ...
  \{ blkDoor₁, blkDoor₂, ublkDoor₁, ... }
```

Now, we focus on tools which can be used to partially verify specifications written in our language.

3.4 Verification Aspects

Different tools may be used for our formalism. Firstly, we focus on the CASL checking part, particularly with CATS, the CASL tool set. Then, we discuss the CWB-NC model-checker concerning the dynamic part, restricted to the basic CCS operators. We do not have yet any tools suitable to the whole specification.

CATS. This tool set[1] is made of several parts: parsing of full CASL syntax, static analysis of full CASL, mixfix analysis, several types of encodings, interfacing with existing theorem provers or rewriting tools, and LaTeX formatting. In a general way, all the tools suitable to CASL could be used on this part of the specification, due to its independence from the other one (CCS agents).

Concerning the access control specification, we mainly use the HOL-CASL parser (version 0.4) [18] to perform parsing, static checking and mixfix analysis. The HOL-CASL enables us to find several syntactic errors, and to partially validate the specification.

[1] http://www.tzi.de/cofi/Tools/CATS.html

CWB-NC. The CWB-NC[2] [5] is an automatic verification tool: the Concurrency WorkBench of the New Century. It provides users with a number of different techniques for specifying and verifying finite-state concurrent systems. The tool combines support for several different system-design notations with decision procedures for a number of refinement relations and for determining if systems satisfy temporal formulas.

First of all, we translate our specification into the CCS input language of the CWB-NC tool. This implies a syntactic and semantic restriction. Concerning the syntax, we remove the value passing of the extended CCS, and the conditional structure *if _ then _*. Moreover, we respect the precise syntax of the tool detailed in [5]. About the semantics, we do not take into account the added rules, *i.e.* the rules describing the behaviour of the CCS operator with algebraic terms. The other inference rules are the same as the ones in CWB-NC.

We use this model checker on the previous access control specification to perform several verification steps: simulation, deadlock checking, and proofs of temporal properties (especially safety and liveness ones) written with the mu-calculus and the CTL operators. The model-checker enables us to verify that the dynamic part of the system reacts correctly.

4 Conclusion and Future Works

In this paper, we have focused on the design of a language which combines the CCS process algebra with the CASL algebraic specification formalism. We have discussed the underlying formal concepts of this language, and have introduced its syntax and its global operational semantics. A real size case study was specified with our formalism. This specification is made of 19 CASL sorts (about 800 lines) and 10 CCS agents (about 150 lines). Existing tools are ever usable to perform partial verification.

Concerning future works, firstly it seems interesting for the specifier to be free to choose his/her preferred specification languages. Indeed, (s)he could prefer a formalism for different reasons: suitability with reference to the problem, existence of development tools, expertise of the specifier. A possible track is to generalize this work to a generic combination in order to satisfy this motivation, and then not staying in a combination case restricted to two fixed languages. Different attempts in this direction are reported in [21, 22].

Another interesting direction corresponds to the further development of the verification aspects. Particularly, we aim at embedding our proposal into languages underlying higher-order logic tools like PVS [12] or Isabelle/HOL [19] to make proofs on the global specification, and not only on independent parts.

[2] http://www.cs.sunysb.edu/˜cwb/

0-7695-1573-8/02/$17.00 (C) 2002 IEEE

The final idea is the extension of the behaviour part to timed CCS [17], so that timed properties could be modelled in a formal and homogeneous manner. Another attempt would be to replace CCS with a timed version of CSP [24] to make the use of multi-way synchronization (and other expressive operators) and of real-time features possible.

References

[1] E. Astesiano, M. Broy, and G. Reggio. Algebraic Specification of Concurrent Systems. In E. Astesiano, B. Krieg-Bruckner, and H.-J. Kreowski, editors, *IFIP WG 1.3 Book on Algebraic Foundations of System Specification*. Springer-Verlag, 1999.

[2] E. Astesiano, M. Cerioli, and G. Reggio. Plugging Data Constructs into Paradigm-Specific Languages: towards an Application to UML. In T. Rus, editor, *Proceedings of AMAST'00*, volume 1816 of *LNCS*, pages 273–292, USA, 2000. Springer-Verlag.

[3] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. In P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–73. Elsevier Science Publishers North-Holland, 1989.

[4] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An Overview of ELAN. In C. Kirchner and H. Kirchner, editors, *Proceedings of the International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*, France, 1998. Elsevier Science.

[5] R. Cleaveland, T. Li, and S. Sims. *The Concurrency Workbench of the New Century (Version 1.2)*. Department of Computer Science, North Carolina State University, 2000.

[6] CoFI. The Common Framework Initiative for Algebraic Specification and Development, electronic archives. Notes and Documents accessible by WWW[3] and FTP[4].

[7] CoFI Language Design Task Group. CASL – The CoFI Algebraic Specification Language – Summary (version 1.0.1). `Documents/CASL/Summary` in [6], March 2001.

[8] CoFI Semantics Task Group. CASL – The CoFI Algebraic Specification Language – Semantics (version 1.0). Note S-9 in [6], 2000.

[9] C. Fischer. How to combine Z with a process algebra. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *Proceedings of ZUM'98*, volume 1493 of *LNCS*, pages 5–23, Germany, 1998. Springer-Verlag.

[10] A. J. Galloway and W. Stoddart. An Operational Semantics for ZCCS. In M. G. H. S. Liu, editor, *Proceedings of ICFEM'97*, pages 272–282. IEEE Computer Society Press, 1997.

[11] H. Garavel and M. Sighireanu. Towards a Second Generation of Formal Description Technique – Rationale for the Design of E-LOTOS. In *Proceedings of FMICS'98*, 1998.

[3] `http://www.brics.dk/Projects/CoFI`
[4] `ftp://ftp.brics.dk/Projects/CoFI`

[12] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial Introduction to PVS. In *Proceedings of WIFT'95*, USA, 1995. Computer Science Laboratory, SRI International.

[13] I. JTC1/SC7/WG14. The E-LOTOS Final Draft International Standard (in balloting). Available at `ftp://ftp.inrialpes.fr/pub/vasy/publications/elotos/elotos-fdis/`, July 2001.

[14] H. Kirchner and C. Ringeissen. Executing CASL Equational Specifications with the ELAN Rewrite Engine. Note T-9 in [6], November 2000.

[15] Y. Ledru. Case Study: Access Control System. IMAG, 1998.

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

[17] F. Moller and C. Tofts. A Temporal Calculus of Communicating Systems. In J. C. M. Baeten and J. W. Klop, editors, *Proceedings of CONCUR'90*, volume 458 of *LNCS*, pages 401–415, The Netherlands, 1990. Springer-Verlag.

[18] T. Mossakowski. CASL: From Semantics to Tools. In S. Graf and M. Schwartzbach, editors, *Proceedings of TACAS'00*, volume 1785 of *LNCS*, pages 93–108, Germany, 2000. Springer-Verlag.

[19] L. C. Paulson. *Isabelle: a Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, New York, USA, 1994.

[20] G. Reggio and L. Repetto. Casl-Chart: A Combination of Statecharts and of the Algebraic Specification Language Casl. In T. Rus, editor, *Proceedings of AMAST'00*, volume 1816 of *LNCS*, pages 243–257, USA, 2000. Springer-Verlag.

[21] G. Salaün, M. Allemand, and C. Attiogbé. A Practical Combination of a Process Algebra with an Algebraic Specification Language. In H. R. Arabnia, editor, *Proceedings of PDPTA'01*, CSREA Press, pages 73–79, Las Vegas, USA, 2001.

[22] G. Salaün, M. Allemand, and C. Attiogbé. Formal Framework for a Generic Combination of a Process Algebra with an Algebraic Specification Language: an Overview. In J. He, editor, *Proceedings of APSEC'01*, IEEE Computer Society Press, pages 299–302, Macau, China, 2001.

[23] G. Salaün, M. Allemand, and C. Attiogbé. A Formalism Combining CCS and CASL. Technical Report 00.14, University of Nantes, December 2000, revised in October 2001. Available at `http://www.sciences.univ-nantes.fr/info/perso/permanents/salaun/papers/rr0014.ps`.

[24] S. Schneider, J. Davies, D. M. Jackson, G. M. Reed, J. N. Reed, and A. W. Roscoe. Timed CSP: Theory and Practice. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Proceedings of Real-Time: Theory in Practice*, volume 600 of *LNCS*, pages 640–675, Germany, 1992. Springer.

[25] H. Treharne and S. Schneider. Using a Process Algebra to Control B OPERATIONS. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proceedings of IFM'99*, pages 437–457, UK, 1999. Springer-Verlag.

0-7695-1573-8/02/$17.00 (C) 2002 IEEE