Synchronous Specialization of Alf for Cyber-Physical Systems

Alessandro Gerlinger Romero
Brazilian National Institute for Space Research, Avenida dos Astrounautas, 1758, 12227-010, São José dos Campos, São Paulo, Brazil.
Email: romgerale@yahoo.com.br

Klaus Schneider
University of Kaiserslautern Computer Science Department, Po box 3049, 67653, Kaiserslautern, Germany.
Email: klaus.schneider@cs.uni-kl.de

Maurício Gonçalves Vieira Ferreira
Brazilian National Institute for Space Research, Avenida dos Astrounautas, 1758, 12227-010, São José dos Campos, São Paulo, Brazil.
Email: mauricio@ccs.inpe.br

***Abstract*—Systems engineers use SysML as a vendor-independent language to model Cyber-Physical Systems. However, SysML does not provide an executable form to define behavior but this is needed to detect critical issues as soon as possible. Alf integrated with SysML can offer some degree of precision. In this paper, we present an Action Language for Foundational UML (Alf) specialization that introduces the synchronous-reactive Model of Computation to SysML, through definition of not explicitly constrained semantics: timing, concurrency, and inter-object communication. The Smart Parking system, a well-known cyber-physical system, was selected to evaluate this specialization. Our initial results show that the proposed specialization does not add complexity to the task of modeling using SysML, and enables concise and precise behavioral definitions.**

***Index Terms*— Alf, CPS, Cyber-Physical Systems, MDA, synchronous-reactive, MoC, system modeling, SysML.**

I. Introduction

Cyber-Physical Systems (CPSs) are an integration of computational and physical processes. Embedded computers and networks monitor and control physical processes with feedback loops where physical processes affect computations and vice versa [17].

According to Cartwright *et. al.* [9], the difficulty in modeling CPSs comes from the diversity of these systems; therefore, the most promising approach to mitigate this problem is to develop expressive and precise modeling languages.

As a result, a large number of languages and formalisms have been proposed to model CPSs [8]. One particular subset of these languages has been established as a technology of choice for specifying, modeling, and verifying real-time embedded applications [4]. This subset is called synchronous languages because it follows the synchronous-reactive Model of Computation (MoC) [17].

This work was supported by the Brazil's Coordination for Enhancement of Higher Education Personnel (CAPES) and German Academic Exchange Service (DAAD).

The synchronous-reactive MoC provides a precise behavioural representation using the fundamental model of time as a sequence of discrete instants, computation and communication executed in zero-time, and parallel composition as a conjunction of behaviors [4]. There is a solid mathematical foundation that supports synchronous-reactive MoC, which allows formal analysis and verification. Languages based on this MoC, like Esterel [6], have been developed and used for safety-critical systems [4][27]. Some of them, like Quartz [29], have been extended for CPSs [3] through introduction of mechanisms to deal with continuous time.

Comparing a system described in the synchronous-reactive MoC against an asynchronous system for dual redundant flight guidance system, Miller *et al* [20] made the following observation: "*the properties themselves are more difficult to state, were weaker than could be achieved in the synchronous case, and required considerable complexity to be added to the model to ensure that even the weakened properties were true*".

Meanwhile, the Object Management Group (OMG) and the International Council on Systems Engineering (INCOSE) are developing the Systems Modeling Language (SysML) [25]; a general-purpose modeling language for systems engineering applications based on the Unified Modeling Language (UML) [23]. SysML has demonstrated a capability for top-down design refinement, but the lack of formal foundations in the SysML results in imprecise behavioural models.

In this paper, we present a specialization to the Action Language for Foundational UML (Alf) [26] for behavioural modeling of CPSs. The hypothesis of this work is that a specialization of Alf according to the synchronous-reactive MoC can be sufficiently expressive to model the discrete behavior of CPSs using SysML. Consequently, adhering to the synchronous-reactive MoC, we will benefit from a solid mathematical foundation [4][6][29].

The remainder of this paper is organized as follows: in Section II, related works are explored briefly; in Section III, we present the initial approach; in Section IV, a case study is presented; in Section V, we briefly discuss the initial approach and the case study; finally, conclusions are shared in the last section.

II. Related Works

There is a large number of research papers about how to formalize semantics for models defined using UML, and consequently, SysML. Hußmann [15] proposes the following classification for approaches concerning structural semantics: (a) naive set-theory, (b) metamodeling, and (c) translation. This classification can be used for the works focused on behavioural semantics.

Extending naive set-theory approach, Graves and Bijan [14] propose one approach where behaviors defined using SysML State Machine Diagrams are axiomatized using type theory.

Alf [26], and the foundational subset for executable UML models (fUML) [24], follows the approach metamodeling because the semantics of behaviors is described operationally using fUML itself. The circularity is broken by the base semantics of fUML, which is specified using first order logic.

Following (c) translation, Bousse *et. al.* [7] define a method to transform a subset of SysML in B method representations; the selected subset of SysML covers behavioural definitions expressed by Alf. Later, the B method representation is proved by a specialized tool. Abdelhalim *et. al.* [1] define a method that receiving State Machine Diagrams and Activity Diagrams (according to fUML) applies a transformation to Communicating Sequential Process (CSP). Afterwards, the CSP representation is verified by a specialized tool.

Benyahia *et. al.* [5] show that fUML, and also Alf, is not directly feasible to safety-critical systems because the MoC defined in the fUML execution model (as it is) is nondeterministic and sequential.

III. Initial Approach

Execution and verification of models is the cornerstone of any Model-Driven Development (MDD). One prominent alternative for MDD is Model-Driven Architecture (MDA) established by OMG [22].

MDA defines three levels of abstraction: (A) Computational Independent Model (CIM) – focuses on the environment of the mission and mission's requirements; (B) Platform Independent Model (PIM) – defines requirements, structure, and behavior for candidate abstract solutions; (C) PSM (Platform Specification Model) – describes concrete solutions. MDA established a large number of specifications, but for this paper the most important is the Alf [26].

Alf is the concrete syntax for the abstract action language defined by fUML [24], a subset of UML [23]; the execution semantics for Alf is given by fUML. According to INCOSE [16], fUML and Alf are MDA pillars for the definition of PIMs.

fUML [24], which defines the semantics for Alf, is designed to support more than one MoC; this is pursued with leaving some semantics elements unconstrained. These elements define aspects of concurrency and inter-object communication which work for simulation, whereas they are not suitable for formal verification. fUML does not define semantics for: (A) timing, (B) concurrency and (C) inter-object communication.

Our initial approach is: given the semantics defined by fUML, we specialize the explicitly unconstrained elements with the purpose of precise definition of models using Alf. In order to do this, we discuss proposed changes in the semantics of fUML. Further, we choose to discuss the semantics in an informal way, and to present a concrete additional language construct for the specialization of Alf. This additional language construct is defined using *Annotation*; according to Alf abstract syntax [26]; it is a way to identify a modification to the behavior of an annotated statement. The applied approach allows us early evaluation of the proposed specialization.

A. Timing

The timing semantics used divides the time scale in a discrete succession of instants; each instant corresponds to one macro-step as defined in the next subsection.

B. Concurrency

Concurrency can be achieved in Alf using two complementary techniques: (A) multiple active objects that, in general, imply the necessity of inter-object communication; or, (B) inside a given definition by the use of the annotation *@parallel*.

Active objects are the source of all behaviors, in a system modeled with UML [22], SysML, fUML, and Alf. An active object is an instance of an active class. An active class must have a *ClassifierBehavior* that defines the class behavior. Each active object is executed independently, and the only way to communicate with other active objects is through signals [23].

One alternative to provide a combination of concurrency and synchrony (where computation and communication are instantaneous) is by using the synchronous-reactive MoC. According to this MoC, a program can be defined by so-called micro and macro steps. Each macro-step is divided into finitely many micro steps, which are all executed in zero time and within the same variable environment (i.e., the ordering of micro steps does not influence the semantics of a model). As a consequence, the values of the variables are uniquely defined for each macro step. Macro steps correspond to reactions of reactive systems, while micro steps correspond with atomic actions, e.g., assignments of the model that implements these reactions [29].

The demarcation of macro steps was introduced by the annotation *@pausable*; it is one of two ways to define demarcation between two macro steps. The second way is the use of *accept* statement from Alf. This annotation is designed to be used with loop constructs (*while, for, do while*), and the semantics is: after each execution of the loop body, it waits for the next macro step. It follows that all concurrent behaviors run in lockstep: they execute the actions inside the loop in zero time, and synchronize before the next iteration.

The annotation *@parallel* can be used to define that all the statements in the block are executed concurrently. The block does not complete execution until all statements complete their execution; i.e., there is an implicit join of the concurrent executions of the statements [26].

C. Inter-Object Communication

Inter-object communication in Alf is performed sending signals (*SendSignalAction*) to other active object [24]. Further, this action is not blocking, i.e., an object sends a signal and continues its execution, it does not wait for a response, or an acknowledgment. A signal is a specification of what can be carried; furthermore, a signal event represents the receipt of a signal instance in an active object [24].

Signals are based on the paradigm of message passing; furthermore, fUML provides a point-to-point (also known as unicast) message pattern [24]. A signal is sent to a receiver (active object) using a reference to it. In contrast, multicasting is required in many safety-critical systems, e.g., fault-tolerance by active redundancy [21]. Multicasting also supports the non-intrusive observation of component interactions by an independent object. Moreover, it enables a better composition.

Multicasting was introduced by an active class called *MessageDispatcher*; it provides the service for multicast message exchange. Instances of this class work as bus transferring instances of signals between previously registered active objects, which generate events in the target active object.

Every signal handled by *MessageDispatcher* has a specific identifiable sender, and zero or more receivers. The set of active objects (receivers) is defined by existence of the reception for that signal. All signals generated in current macro-step are available instantaneously. Moreover, signals not used during a macro step are lost. It is possible to receive signals individually or as a set. Receiving a set of signals is important for those active objects that need to process all signals sent for it in current macro-step.

IV. CASE STUDY

A case study was developed to evaluate our initial approach. Points discussed above were applied to model part of a system called *Smart Parking*. The *SmartParking* has chosen for three respective reasons: (1) it is a real-world Cyber-Physical System; (2) it can be modeled as a discrete system [13]; (3) Geng and Cassandras [13] provide a detailed concrete solution.

In accordance with [11][28], the case study is defined using MDA. The case study focus on aspects related to computation and communication in the PIM abstraction level, one option to cover the control aspect is presented in [13].

A. Mission Context and Requirements

Mission context and mission requirements were gathered and modeled in a SysML CIM Model. The mission is summarized below.

A user, inside a vehicle, shall be able to request a parking space. The request for a parking space shall be evaluated considering two constraints given by the user: (a) maximum distance from current position, and (b) maximum cost that the user wants to pay.

The user shall receive a response indicating the best parking space that satisfies the imposed constraints. The user shall be able to accept or reject this response.

The user shall be informed about where is the parking space reserved for him, as well as, about the availability of all the parking spaces up to 10 meters from his current position.

The vehicle shall be able to send its current position. The vehicle shall be detected when it arrives at a parking space, and when it departures from a parking space.

B. An Abstract Solution

Fig. 1 shows the Block Definition Diagram (BDD) for an abstract solution, which is compatible with the concrete solution defined in [13]. The *SmartParking* system was decomposed in three main parts: *SmartParkingEnablerDevice*, *SmartParkingAllocationCenter*, and *Spot*; all of them are active classes.

The connections between these elements are not static; therefore, they are not presented in Fig. 1 as associations. The connections are showed in the Internal Block Diagram (IBD) presented in Fig. 2. In contrast to associations, which specify links between any instances of the associated classifiers, connectors specify links between instances playing the connected parts only [23]. The inter-object communication is provided by the multicast message exchange service (*MessageDispatcher*); further, each active object has a reference to the same instance of *MessageDispatcher*.

SmartParkingEnablerDevice models a device inside the vehicle. It receives *Position* from vehicle, and has a *UserInterface* (both interactions with the environment depicted left-up corner in the Fig. 2.). Each *Vehicle* has a corresponding *SmartParkingEnablerDevice* active object. The abstraction used in this case study makes internal structure of this component irrelevant. It, as well as other components, could be modeled later as software, hardware or a composition of both; e.g., *SmartParkingEnablerDevice* could be implemented as software in a smartphone [13].

Each parking space managed by the system is an active object *Spot*. Each *Spot* has two interactions with the environment; (a) detecting that a vehicle arrived at a *Spot* (*VehiclePresenceSensor)*; and, (b) indicating for a user what is the current state of the *Spot*, and which one is reserved for him (*LightsActuator*).

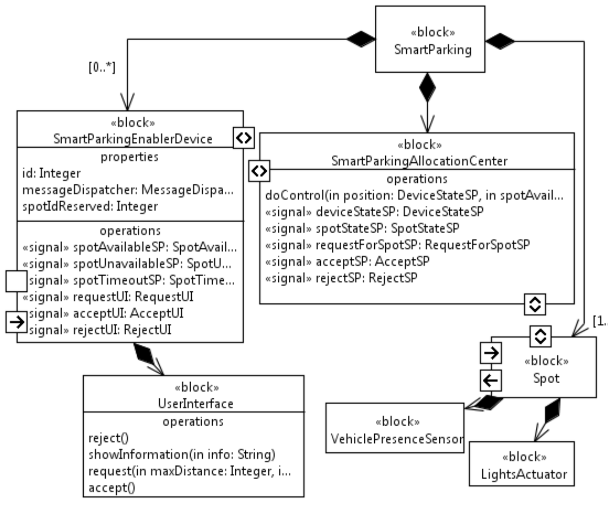

«block»
SmartParking
[0..*]
«block»
SmartParkingEnablerDevice
properties
id: Integer
messageDispatcher: MessageDispa...
spotIdReserved: Integer
operations
«signal» spotAvailableSP: SpotAvail...
«signal» spotUnavailableSP: SpotU...
signal» spotTimeoutSP: SpotTime...
«signal» requestUI: RequestUI
signal» acceptUI: AcceptUI
signal» rejectUI: RejectUI
«block»
SmartParkingAllocationCenter
operations
doControl(in position: DeviceStateSP, in spotAvail...
«signal» deviceStateSP: DeviceStateSP
«signal» spotStateSP: SpotStateSP
«signal» requestForSpotSP: RequestForSpotSP
«signal» acceptSP: AcceptSP
«signal» rejectSP: RejectSP
[1..*]
«block»
Spot
«block»
UserInterface
operations
reject()
showInformation(in info: String)
request(in maxDistance: Integer, i...
accept()
«block»
VehiclePresenceSensor
«block»
LightsActuator

Fig. 1. BBD System components (PIM level).

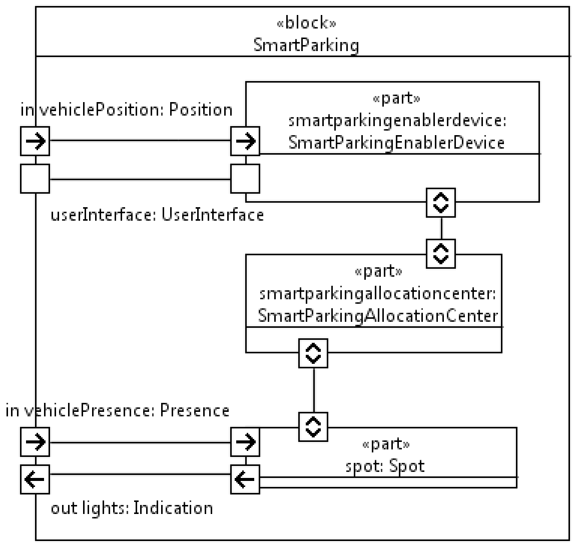

«block»
SmartParking
in vehiclePosition: Position
«part»
smartparkingenablerdevice:
SmartParkingEnablerDevice
userInterface: UserInterface
«part»
smartparkingallocationcenter:
SmartParkingAllocationCenter
in vehiclePresence: Presence
«part»
spot: Spot
out lights: Indication

Fig. 2. IBD System abstract solution (PIM level).

Spot and *SmartParkingEnablerDevice* (plant) are managed by the block *SmartParkingAllocationCenter* (controller). In this case study, there is only one active object from this block, which in each macro step is responsible for: (a) gathering system state and events; and (b) determining the control output.

From the viewpoint of Discrete Event Systems (DES) control, considering signals handled by *SmartParkingAllocationCenter*, the system can be described as follows:

$$X(t) = \{D(t), P(t)\} \tag{1}$$

$$E = ECent \cup ED \cup ESpot \tag{2}$$

$$ECent = \{RequestForSpotSP, AcceptSP, RejectSP\} \tag{3}$$
$$ED = \{SpotAvailableSP, SpotUnavailableSP, SpotTimeoutSP\} \tag{4}$$
$$ESpot = \{AllocatedSP, UnallocatedSP\} \tag{5}$$

$$X(t+1) = f(X(t), U(t), W(t)) \tag{6}$$

where:
(1) defines the discrete state space *X(t)* - composed by *D(t)* = {k ∈ *Natural*: *SmartParkingEnablerDevice* k in the system} determined in each macro step by signal events of the signal *DeviceStateSP*; and, *P(t)* = {k ∈ *Natural*: *Spot* k in the system} determined in each macro step by signal events of the signal *SpotStateSP*;
(2) is the discrete event set, which is composed by signals: (3) received from *SmartParkingEnablerDevice* (ECent)*;* (4) sent to *SmartParkingEnablerDevice* (ED); and, (5) sent to *Spot* (ESpot);
(6) defines the evolution of the system over time, which is the state X in the next macro step (t + 1) is defined by the state, events and control in the current macro step t; *X(t)* defines the state in the instant *t*; *U(t)* = {k ∈ *Natural*, i ∈ *ED* or i ∈ *ESpot*: *instance* i_k } is the set of control signals determined in each instant *t* by instances of the signals defined in the sets *ED* and *ESpot*; and, *W(t)* = {k ∈ *Natural*, i ∈ *ECent*: *instance* i_k } is the set of signals (events) determined in each intant *t* by instances of signals defined in the set *ESpot.*

Fig. 3. shows that the Alf *ClassifierBehavior* of the *SmartParkingEnablerDevice* has two concurrent infinite loops.

The first infinite loop depicted in Fig. 3. is annotated with *@pausable*, which means that it sends the current state of device; thereupon, it waits for the next macro step (synchronization point, before next iteration). The current state is composed by the actual position and the state of current reservation, and is represented by an instance of the signal *DeviceStateSP*. Each active object sends this signal in each macro step using an instance of *MessageDispatcher* that is responsible for delivering a copy of these messages to every registered active object that has a reception for this signal.

The second infinite loop defines the expected reactions of the device for events received from *UserInterface* and from *SmartParkingAllocationCenter*. It starts with an *accept* statement, which blocks execution (possible during many macro steps) until the expected event occur. Subsequently, it uses the same mechanisms described above to send signals for other active objects. Moreover, it uses a compound *accept* statements that determines which block will be activated based on the type of the signal received from *UserInterface* and from *SmartParkingAllocationCenter*.

SmartParkingAllocationCenter behavior is showed in Fig. 4. It has an infinite loop annotated with *@pausable* that defines a synchronization point in the end of each execution of the loop body. The loop body starts with five concurrent *accept* statements, which means that it waits until no more signals of these types can be generated; later, it applies the

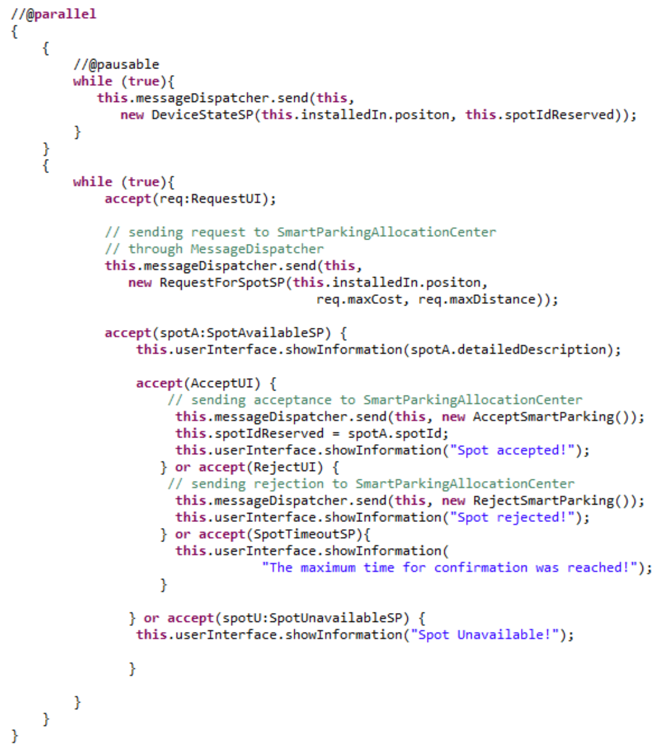
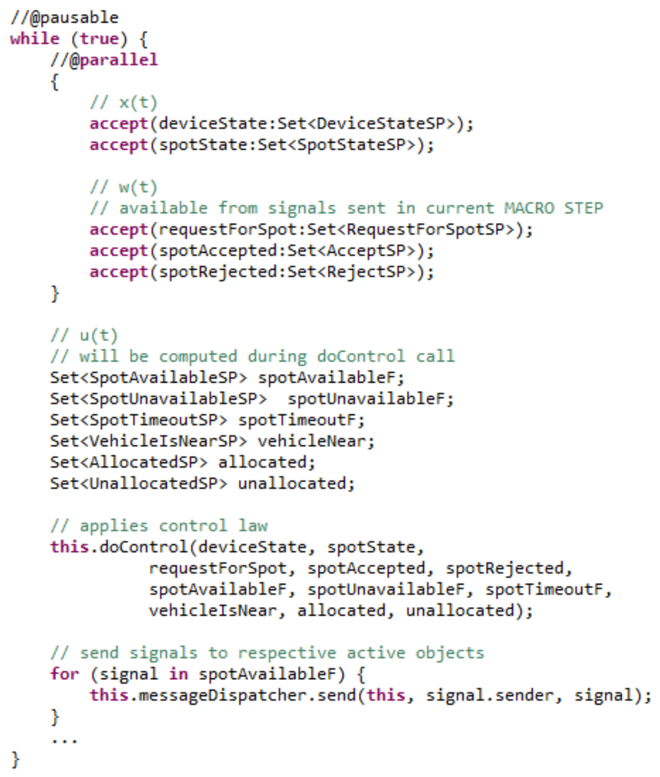

```
//@parallel
{
    {
        //@pausable
        while (true){
           this.messageDispatcher.send(this,
              new DeviceStateSP(this.installedIn.positon, this.spotIdReserved));
        }
    }
    {
        while (true){
            accept(req:RequestUI);

            // sending request to SmartParkingAllocationCenter
            // through MessageDispatcher
            this.messageDispatcher.send(this,
               new RequestForSpotSP(this.installedIn.positon,
                                       req.maxCost, req.maxDistance));

            accept(spotA:SpotAvailableSP) {
                this.userInterface.showInformation(spotA.detailedDescription);

                accept(AcceptUI) {
                    // sending acceptance to SmartParkingAllocationCenter
                     this.messageDispatcher.send(this, new AcceptSmartParking());
                     this.spotIdReserved = spotA.spotId;
                     this.userInterface.showInformation("Spot accepted!");
                   } or accept(RejectUI) {
                    // sending rejection to SmartParkingAllocationCenter
                     this.messageDispatcher.send(this, new RejectSmartParking());
                     this.userInterface.showInformation("Spot rejected!");
                   } or accept(SpotTimeoutSP){
                     this.userInterface.showInformation(
                               "The maximum time for confirmation was reached!");
                   }

              } or accept(spotU:SpotUnavailableSP) {
                this.userInterface.showInformation("Spot Unavailable!");

              }

        }
    }
}
```

Fig. 3. Alf *ClassifierBehavior* of *SmartParkingEnablerDevice*.



```
//@pausable
while (true) {
    //@parallel
    {
        // x(t)
        accept(deviceState:Set<DeviceStateSP>);
        accept(spotState:Set<SpotStateSP>);

        // w(t)
        // available from signals sent in current MACRO STEP
        accept(requestForSpot:Set<RequestForSpotSP>);
        accept(spotAccepted:Set<AcceptSP>);
        accept(spotRejected:Set<RejectSP>);
    }

    // u(t)
    // will be computed during doControl call
    Set<SpotAvailableSP> spotAvailableF;
    Set<SpotUnavailableSP>  spotUnavailableF;
    Set<SpotTimeoutSP> spotTimeoutF;
    Set<VehicleIsNearSP> vehicleNear;
    Set<AllocatedSP> allocated;
    Set<UnallocatedSP> unallocated;

    // applies control law
    this.doControl(deviceState, spotState,
              requestForSpot, spotAccepted, spotRejected,
              spotAvailableF, spotUnavailableF, spotTimeoutF,
              vehicleIsNear, allocated, unallocated);

    // send signals to respective active objects
    for (signal in spotAvailableF) {
        this.messageDispatcher.send(this, signal.sender, signal);
    }
    ...
}
```

Fig. 4. Alf *ClassifierBehavior* of *SmartParkingAllocationCenter*.

control law, and sends the response for other active objects (*SmartParkingEnablerDevice* and *Spots*) using the mechanism described above.

The Alf *ClassifierBehavior* of the *Spot* has the same organization that *SmartParkingEnablerDevice*. There are two concurrent infinite loops: one sending signals about its state (with synchronization point defined using *@pausable*), and, one defining reactions for the received events from *VehiclePresenceSensor* and from *SmartParkingAllocationCenter*.

V. Discussion

The case study defines one abstract solution (PIM) for the mission that was modeled to explore: concurrency, synchronization, and multicast messages. The solution is neither complete nor optimized, e.g., signals can be removed by a centralized version of the state of the system. A tradeoff could be evaluated taking into account an objective function defined at CIM level, e.g., considering the analysis of the messages (communication) during macro steps. In addition, the abstract solution has an important difference compared to the solution presented in [13]: there are no queues. This is a consequence of the synchronous-reactive MoC; all signals are received and processed in the same macro step. The *SmartParkingEnablerDevice* does not have the state "Waiting for Assignment" [13] because, given a macro step, the system state is gathered instantaneously; afterwards, the control law is applied; and, all active objects in *SmartParking* immediately receive an adequate response.

From the viewpoint of DES control [10], the case study satisfies the cornerstone properties: (a) its state space is a discrete set, as defined in (1); and, (b) the state transition mechanism is event-driven, which means that the state can only change as a result of asynchronously occurring instantaneous events over time [10]. Apart from that, the second property has a time window to occur, during a macro step. In the case study, it is mandatory that many events occur in the same macro step, and the resulting state transition reflects the occurrence of all. However, some combinations of signals in the same macro step is not allowed, e.g., if a naive device sends in a given macro step one signal for requesting a spot, and one signal for acceptance, the last one will be lost.

Concerning modeling, StateMachines and State Machine Diagrams are commonly used for modeling state-dependent behavior. A variation of these diagrams is used to express state-dependent behavior in [13]. However, UML, fUML, SysML, and Alf do not define precise semantics for state machines [12][30]. This is ratified by Alf, which states that a normative semantic integration of state machines with Alf will be formalized later as a part of future standards [26]. Indeed, environments of synchronous languages offer tools to visualize the resulting automata from a given text representation [6], e.g. Fig. 3. can be automatically transformed in a StateMachine Diagram. Languages have been developed to conciliate precise semantics and automata visual modeling as [2][18].

The nondeterminism in the fUML MoC, which was recognized by Benyahia *et. al.* [5], can be removed using the proposed specialization. In fact, the proposed specialization adheres the idea of introducing synchronous-reactive MoC during early stages of a system development [4]. It avoids asynchronous complexity in early stages of system modeling, analyzing, and verification. Furthermore, the synchronous-reactive MoC enables abstract solutions to be synthetized [27] in a concrete solution using Globally Asynchronous Locally Synchronous architectures (GALS) [20], or Physically Asynchronous Locally Synchronous architectures (PALS) [19]. The initial approach presented here provides rather a starting point than a complete result. It defines informally the semantics for two complementary constructs for Alf that together can transform Alf in a synchronous action language; however, the changes needed in the fUML execution model to support it must be defined, and the points about nondeterminism stated in [5] have to be addressed.

CPS is about the intersection of the computation, communication, and control [17]. The initial approach focuses on the computational and communicational aspects of CPSs, and it can be composed with control. The case study shows that our initial approach can transfer solid mathematical foundation from synchronous languages to SysML executable models. We consider this step, as an intermediary step, before a formal verification of executable discrete SysML models.

VI. Conclusions

This paper shows the initial results of our research that has the following basic hypothesis: a specialization of Alf according to synchronous-reactive MoC can be sufficiently expressive to model the discrete behavior of CPSs systems using SysML.

These results show that the proposed specialization does not add complexity to the task of modeling using SysML, and enables concise and precise behavior definition.

We believe that specializing well-known vendor-independent specifications (Alf and SysML) can provide an understandable set of languages for modeling, analyzing and verification of CPSs. Moreover, such a set of languages can enable formal verification for discrete parts of CPSs.

REFERENCES

[1] Abdelhalim, I.; Schneider, S.; Treharne, H. (2012). An Optimization Approach for Effective Formalized fUML Model Checking. In Proc. SEFM2012 Proceeding of the 10th International Conference on Software Engineering and Formal methods, 2012, pg. 248-262.

[2] Andre, C. (1995). Synccharts: a visual representation of reactive behaviors. Technical Report RR 95–52, I3S, Sophia-Antipolis, France, October 1995.

[3] Bauer, K. (2012). A New Modelling Language for Cyber-physical Systems. PhD thesis, Department of Computer Science, University of Kaiserslautern, Germany, Kaiserslautern, Germany, January 2012.

[4] Benveniste, A.; Caspi, P.; Edwards, S.; Halbwachs, N.; Guernic, P.; Simone, R. (2003). The synchronous languages twelve years later. Proceedings of the IEEE, 91(1):64–83, 2003.

[5] Benyahia, A.;Cuccuru, A.; Taha, S.; Terrier, F.; Boulanger, F.; Gérard, S. (2010). Extending the Standard Execution Model of UML for Real-Time Systems. In Proc. DIPES/BICC, 2010, pp.43-54.

[6] Berry, G. (2000). The Esterel v5 Language Primer: version:5.91. France. Available at: < http://francois.touchard.perso.esil.univmed.fr/3/esterel/primer.pdf>. Access date: 14.Apr.2013.

[7] Bousse, E.; Mentré, D. Combemale, B.; Baudry, B.; Katsuragi, T. (2012) Aligning SysML with the B Method to Provide V&V for Systems Engineering. Proc. Of 12th Model-Driven Engineering, Verification, and Validation 2012.

[8] Carloni, L.; Benedetto, D.; Passerone, R.; Pinto, A.; Sangiovanni-Vincentelli, A. (2004). Modeling Techniques, Programming Languages and Design Toolsets for Hybrid Systems. Project IST-2001-38314 COLUMBUS. Design of Embedded Controllers for Safety Critical Systems. WPHS: Hybrid System Modeling.

[9] Cartwright, R.; Kelly, K.; Koushanfar, F.; Taha, W. (2006). Model-Centric Cyber-Physical Computing. In proceedings … NSF Workshop on Cyber-Physical Systems, 2006, Austin, Texas: USA.

[10] Cassandras, C.; Lafortune, S. (2010). Introduction to Discrete Event Systems. Second Edition, Springer Science+Business Media, New York, 2010.

[11] Cloutier, R. (2006) MDA for systems engineering – Why should we care? USA. Available at: <http://www.calimar.com/Papers/Model%20Driven%20Architecture%20for%20SE-Why%20Care.pdf>. Access date: 25.Jun.2010.

[12] Fecher, H.; Schönborn, J.; Kyas, M.; Roever, W. (2005). 29 New Unclarities in the Semantics of UML 2.0 State Machines. In Proceedings of the International Conference on Formal Engineering Methods, LNCS 3785, Berlin/Heidelberg, Germany, Springer-Verlag, 2005, pg 52-65.

[13] Geng, Y.; Cassandras, C. (2011). "A New "Smart Parking" System Based on Optimal Resource Allocation and Reservations", Proc. of 14th IEEE Intelligent Transportation Systems Conf., pp. 979-984, Nov. 2011.

[14] Graves, H.; Bijan, Y. (2011). Using formal methods with SysML in aerospace design and engineering. Journal Annals of Mathematics and Artifical Intelligence. Volume 63, Issue1, September, 2011. pg 53-102.

[15] Hußmann, H. (2002). Loose semantics for UML, OCL, in: Proceedings 6th World Conference on Integrated Design and Process Technology, IDPT 2002, June, Society for Design and Process Science, 2002.

[16] International Council on Systems Engineering (INCOSE). (2008). Survey of Model-Based Systems Engineering (MBSE) methodologies. USA: INCOSE, Seattle, 2008. 80 p. Available at: <http://www.incose.org/productspubs/pdf/techdata/MTTC/MBSE_Methodology_Survey_2008-0610_RevB-JAE2.pdf>. Access date: 25 jun. 2010.

[17] Lee, E.; Seshia, S. (2011). Introduction to Embedded Systems - A Cyber-Physical Systems Approach. http://leeseshia.org/, 2011. ISBN 978-0-557-70857-4.

[18] Maraninchi, F. (1991). The Argos Language: Graphical Representation of Automata and Description of Reactive Systems. In International Conference on Visual Languages, Kobi, Japan, IEEE Computer Society, 1991.

[19] Meseguer, J.; Ölveczky, C. (2010). Formalization and Correctness of the PALS Architectural Pattern for Distributed Real-Time Systems. Formal Methods and Software Engineering. Lecture Notes in Computer Science, 2010, Volume 00206447/2010, 303-320, DOI: 10.1007/978-3-642-16901-4_21.

[20] Miller, P.; Whalen, M.; Obrien, D.; Heimdahl, M.; Joshi, A. (2005). A methodology for the design and verification of globally asynchronous/locally synchronous architectures. NASA Contractor Report NASA/CR-2005-213912.

[21] Obermaisser, R.; Kopetz, H. (2009). Genesys – A candidate for an ARTEMIS Cross-Domain Reference Architecture for Embedded Systems. 2009. Available at: <http://www.genesys-platform.eu/genesys_book.pdf> Access date: 17.May.2011.

[22] Object Management Group (OMG). (2003). Model-Driven Architecture. USA: OMG, 2003. Available at: <http://www.omg.org/mda>. Acesso em: 17 may. 2009.

[23] Object Management Group (OMG). (2011). Unified Modeling Language Superstructure: Version: 2.4.1. USA: OMG, 2011. Available at: <http://www.omg.org/spec/UML/2.4.1/>. Access date: 14.Apr.2013.

[24] Object Management Group (OMG). (2012). Semantics of a Foundational Subset for Executable UML Models: Version 1.1 RTF Beta. USA: OMG, 2012. Available at: <http://www.omg.org/spec/FUML/>. Access date: 24.Apr.2013.

[25] Object Management Group (OMG). (2012). Systems Modeling Language: Version: 1.3. USA: OMG, 2012. Available at: < http://www.omgsysml.org/>. Access date: 27.Apr.2013.

[26] Object Management Group (OMG). (2013). Concrete Syntax for UML Action Language (Action Language for Foundational UML - ALF): Version: 1.0.1 - Beta. USA: OMG, 2013. Available at: <http://www.omg.org/spec/ALF/>. Access date: 27.Apr.2013.

[27] Potop-Butucaru, D.; Caillaud, B. (2007). Correct-by-Construction Asynchronous Implementation of Modular Synchronous Specifications. Journal Fundamenta Informaticae - The Fourth Special Issue on Applications of Concurrency to System Design (ACSD05) archive Volume 78 Issue 1, September 2007 Pages 131-159.

[28] Romero, A. G.; Ferreira, M. G. V. (2012). An Approach to Model-Driven Architecture applied to Hybrid Systems In: SpaceOps 2012, 2012, Stockholm. 12th International Conference on Space Operations. Stockholm: AIAA, 2012. Available at: <http://spaceops2012.com/proceedings/documents/id1290620-Paper-003.pdf>. Access date: 24.Apr.2013.

[29] Schneider, K. (2009). The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December, 2009.

[30] Von der Beeck, M. (1994). A Comparison of Statechart Variants. In H. Langmaack, W.-P. de Roever, J. Vytopil (eds.): Formal Techniques in Real-Time and Fault-Tolerant Systems. Berlin: Springer. 128-148.