Des Autom Embed Syst (2015) 19:345–366
DOI 10.1007/s10617-015-9164-y

# Cyber-physical systems design: transition from functional to architectural models

**Rosane Fátima Passarini**[1] · **Jean-Marie Farines**[2] · **João M. Fernandes**[3] · **Leandro Buss Becker**[2]

Received: 28 February 2014 / Accepted: 15 April 2015 / Published online: 30 May 2015

**Abstract** Normally, the design process of Cyber-Physical Systems (CPSs) starts with the creation of functional models that are used for simulation purposes. However, most of the time such models are not directly reused for the design of the architecture of the target CPS. As a consequence, more efforts than strictly necessary are spent during the CPS architecture design phase. This paper presents an approach called *Assisted Transformation of Models (AST)*, which aims at transforming functional (simulation) models designed in the Simulink environment into architectural models represented in the Architecture Analysis and Design Language. Using AST, designers can perform a smooth transition between these two design phases, with an additional advantage of assuring the coupling between functional and architectural models. The use and benefits of AST are exemplified in the paper in a study devoted to for the design of a typical CPS: an Unmanned Aerial Vehicle.

**Keywords** Model Transformation · Functional models · Software architecture · Simulink · AADL

✉ Rosane Fátima Passarini
rosane@utfpr.edu.br

Jean-Marie Farines
j.m.farines@ufsc.br

João M. Fernandes
jmf@di.uminho.pt

Leandro Buss Becker
leandro.becker@ufsc.br

[1] Universidade Tecnológica Federal do Paraná (UTFPR), Rua Cristo Rei, 19, Toledo, PR 85902-490, Brazil

[2] Universidade Federal de Santa Catarina (UFSC), DAS/CTC/UFSC, PO Box 476, Florianópolis, SC 88040-900, Brazil

[3] Department of Informática / Centro ALGORITMI, Escola de Engenharia, Universidade do Minho, 4710-057 Braga, Portugal

## 1 Introduction

The denomination Cyber-Physical System (CPS) [13] is commonly used to represent an electromechanical device being controlled by a computer-based system. Examples of CPSs include robots, aircrafts, smart grids, and others (this paper presents the design of an Unmanned Aerial Vehicle (UAV), which is a typical CPS). Given the multidisciplinary nature of CPSs, they must be designed using different kinds of models to represent their mechanical and electrical structure plus the computational ("cyber") part.

The design of the CPS architecture normally starts in the latter phases of the development process. Prior to that, designers carefully model the system kinematics and the control algorithms that will be adopted to rule the CPS behavior. Typically, simulations are performed to properly adjust the behavior of the control algorithms. For instance, the mathematical model that represents the kinematics of the UAV under design in a related project and a suitable algorithm for controlling such UAV is presented in [5].

The Simulink tool [14] is usually adopted for modeling and simulating CPS, addressing the design of the physical system kinematic and the control algorithms. Even though this tool offers the possibility of code generation, it has a weak support for expressing and generating the CPS architecture, as discussed in [8]. When considering a CPS design, generically, the same features and drawbacks related to Simulink also apply to other CPS-related simulation tools such as LabVIEW, Scade, Scilab, Ptolemy, and others.

On the other hand, architecture description languages like the Architecture Analysis and Design Language (AADL) [6] are very appropriate to represent the CPS architecture. Those languages allow the specification of the software architecture in detail, also providing means to express the hardware platform and its association with the software components. According to [6], functional models (from Simulink, Scade, etc) can be used to complement AADL components characterization by providing their functionalities, that is, the algorithms that characterize the system behavior.

The problem under investigation in the present work is how to properly address the design of the CPS architecture using the functional model as a starting point. Traditionally, this has been viewed as a deployment problem that addresses how to allocate a certain functional model into a model representing the CPS architecture and its embedded platform. This requires two models to be designed separately and then combined. For instance, a similar philosophy is behind UML and its MARTE profile [19].

In this paper we address how to generate in an automated way the architectural model taking as input the functional model. This kind of approach avoids creating possible decoupled functional and architectural models [16]. Besides, it promotes a truly "model-driven" design approach, as the former model is mapped into the latter one using a transformation tool. More specifically, the present work suggests how to relate elements of a Simulink model with elements of an AADL model. While the elements of the Simulink model target functionalities, the elements of the AADL model represent a suitable structure and target platform for incorporating those functionalities.

Since both models adopt different semantics, we consider important to have some kind of user intervention in the transformation process. Therefore the proposed approach is called *ASsisted Transformation of models* (AST). It consists of metamodels for the source and target languages, a set of marks (similar to stereotypes), and mapping rules. Besides, it also provides a tool support to automate the model transformations.

The AST approach is part of what we consider to be an adequate model-based design method for CPSs. An overview of such a method is presented in Sect. 2. Section 3 presents some related works and compares them with AST. Section 4 provides an overview of AST and Sect. 5 is devoted for detailing its transformation engine. Section 6 presents a case study that shows the application of AST, highlighting its benefits and limitations. Finally, Sect. 7 presents our conclusions and addresses the future works.

## 2 Overview of the adopted development method

This section provides an overview of the model-based development method for CPS design followed in this work. Such method is based on a previous work from some of the present authors (see [3]). It has evolved along the last years, specially addressing the transformation of functional to architectural specification, which is the focus of the present paper. Due to space constraints, this section is restricted to an overview of such a method and its four suggested design steps, which are: (i) system requirements definition; (ii) preliminary design; (iii) detailed design, and (iv) implementation. Figure 1 depicts the four steps, including the resulting actions (inside the blocks) and the provided outputs.

An important aspect to be highlighted is that this method suggests adopting different modeling languages to represent systems functionalities and architecture. The reason therefore is related with the tools currently available to perform simulations of the system functionalities. CPS designers normally prefer using tools that support mathematical modeling and that include simulation capacity, like for example Simulink, LabVIEW, Scilab, and Ptolemy. However, as discussed in [8], such tools are not appropriate to represent the system architecture. For this reason, the adopted development method suggests using a different modeling language to represent the system architecture.

### 2.1 Definition of the system requirements

The first step of the method is eliciting the system requirements from the sources (typically the stakeholders). This is a non-trivial step to be conducted, since it implies discovering and unveiling the needs of the users with respect to the system being built. There are many techniques that can be used in this step, like interviews, task analysis, domain analysis, introspection, brainstorming, observation, personas. A good survey about elicitation techniques can be found at [22]. As usual in these situations, there are no hard rules to decide which techniques to use. The techniques to be used depend on the judicious evaluation made by the engineer. The end result of this step is a specification of the user requirements, both functional and non-functional, written in natural language (e.g., English). This specification serves mainly as a communication medium between the users and the project members.

When the requirements are considered to be completed, the development team can produce a use case diagram. By constructing this diagram, the development team needs to define two main issues:

1. The actors that the systems interacts with. i.e., the environment or context of the system.
2. The use cases provided by the system, i.e., the main functionalities available to its users.

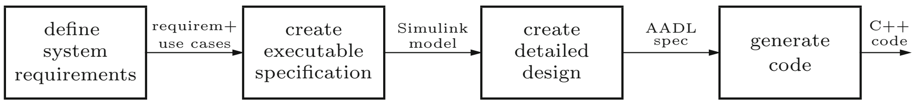

**Fig. 1** Main activities and artifacts of our method to develop CPSs

### 2.2 Preliminary design

The next step is to create a preliminary design for the CPS, focusing on obtaining a simulatable specification. For that purpose, it is proposed that the development team makes use of a modeling language that allows the model to be simulated – both the controller and the cyber environment. An additional interesting feature would be to allow structuring the CPS in terms of components and subsystems. Moreover, the language could also aid the design team to model the possible operation modes of the CPS under design.

To support this design step, we rely on the Simulink tool, which provides a block-diagram language to structure the system being modeled. Functional specification can be represented using block diagrams (using blocks either from a library or user-defined), pieces of Matlab or C source code (used to fulfill user-defined blocks), and state machines. Although not explored within the present work, other tools (and languages) could be used throughout this design step.

### 2.3 Detailed design

Creating a detailed design is mandatory in any engineering project, and it is not different while designing a CPS. At this step, the design team must decide about the allocation of the functionalities into processes and threads, and also reason about the deployment of such tasks into a target platform, which by the way must also be defined in this design step. Moreover, the different constraints related with the system implementation must be taken into account, like for instance timing and energy consumption restrictions.

The use of different modeling languages in the previous and this design steps should allow the development team to gradually change the system representation written in one (more abstract, more informal) language into another (more concrete, more rigorous) representation. The resulting model should have enough details to make it suitable for the code generation phase. Additionally, such model should also be suitable for model-based analysis, so that possible design mistakes could be detected and properly corrected before the code is generated.

AADL was chosen by the present work to be used in this step because it allows expressing in detail the software organization and its target platform. Besides, AADL contains adequate tool support to perform various types of model-based analysis and a proper abstraction level to allow its implementation in a given programming language.

### 2.4 Implementation

For the implementation phase to be properly conducted, the model resulting from the previous design step should have enough details so that generating code from it becomes straightforward, i.e., that programmers or code generation software might be able to interpret it and generate the respective program code in a given target language. For instance, the Ocarina tool [11] can perform automatic code generation from an AADL model to C/C++ or ADA languages. Finally, although this design step could cover hardware/software co-design, this topic is left for a future work.

## 3 Related works

This section presents some works related with the design of embedded systems and CPSs that use AADL and Simulink and that also suggest performing model transformations.

Raghav et al. [16] show how to obtain a preliminary AADL model using information extracted from a Simulink functional model. Such approach requires the use of the System Description and Analysis Language (SADL) as an intermediate language. The observed problem is that the mapping from SADL to AADL has some limitations, for instance, it does not properly map ports and data. It also lacks support for the mapping of other hardware components besides the processor, making it difficult to design a detailed architectural model that is close to the final system. Furthermore, SADL does not support behavior specification using state machines [2], making it hard to map Simulink models that use stateflow diagrams.

The approach presented in [4] suggests the manual integration of functional components designed with Simulink and SCADE tools with the system components represented in an AADL architectural model. Its main goal is the generation of the complete source code of the application from the models. As this approach does not apply model transformation, Simulink or SCADE models must be manually integrated with the AADL architecture model. In any case, the overall results from this work served as inspiration for the Simulink-to-AADL mappings proposed by the present work.

The Polychrony tool presented in [9] was integrated by [21] into the Reference Technology Platform (RTP) of the project named CESAR (Cost-efficient methods and processes for safety relevant embedded systems) [20], serving as a cooperative framework for architecture modeling and exploration. It allows importing high-level Simulink (functional) models and AADL (architectural) specifications. The importing feature is currently implemented for two different transformations, namely Simulink-to-Signal and Signal-to-AADL. Signal [12] is a language for specification and programming of embedded real-time critical applications. Interfaces are required and need to be manually implemented. The composition of Simulink and AADL models depends, therefore, on the system designers to implement such Signal interfaces, making it difficult to maintain and to validate the resulting model.

As far as we are aware of, there is no definitive solution for the problem of mapping Simulink models into AADL models. Most related works provide "links" between them, not mapping (in the sense of a model transformation). None of the works that cover mapping [16,21] deals with behavior. With respect to the structural mappings, the analyzed works make use of intermediate languages, which somehow limits the proposed mappings.

Recently a new initiative from the Software Engineering Institute (SEI), with similar ideas as our approach, was presented. They are developing plugins for OSATE2 that aims at importing models generated in Simulink or Scade to generate the skeleton of an AADL model. Such plugins are being developed under the project SAVI (System Architecture Virtual Integration Program) [1]. However, there is no information available about such plugins besides their preliminary source code,[1] making it difficult for us to judge on their pros and cons. Nevertheless, it was possible to observe in the *importer.simulink* plugin source code that it focuses on the identification of architectural problems related with data sharing. Besides, such plugin does not support the mapping of operation modes, which is tackled in our proposal.

## 4 Assisted transformation of models

This section presents the proposed *Assisted Transformation of models (AST)*. It aims at transforming Simulink models into AADL models, which allows performing the transition from functional to architectural modeling. This proposal contributes to the model-based

[1] https://github.com/osate/osate2-plugins.

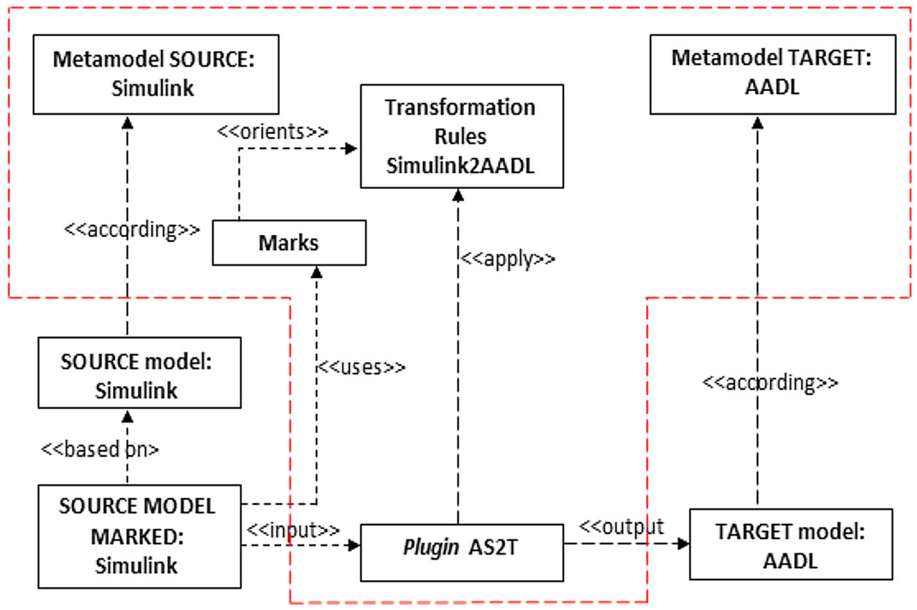

**Fig. 2** Process structure of the assisted transformation of models

development method for CPS presented in Sect. 2. More specifically, it allows the transition from the second phase (preliminary design) to the third phase (detailed design).

Figure 2 provides an overview of the artifacts used in our proposal. It basically consists of a transformation engine that receives as input (source) a Simulink model (an *.mdl* file) and outputs an AADL model (an *.aadl* file). The core of AST is the correlation between Simulink and AADL models. This core was developed along this work and gave origin to the so-called *transformation rules*. Such rules are used in the transformation engine named AS2T plugin that was developed in this work. Metamodels for the input and output languages also created in this work are used to support the proposed transformation. AST can be classified as an approach that performs Model-to-Model (M2M) transformations [15].

## 4.1 AST metamodels

Metamodels are very important elements in the model transformation process, since they define the structure, syntax, and restrictions related to a family of models. They are typically represented as UML class diagrams.

As there is no official version of the Simulink metamodel, and the metamodels available in the related works lack information required by AST, we created a new Simulink metamodel. It provides a compact description of the structural aspects of a Simulink model.

An overview of the proposed Simulink metamodel is presented in Fig. 3. As it can be observed, every Simulink model has a *Model* component, serving as the model root. This *Model* component is composed of a unique *System* type component, which encapsulates a Simulink model and serves as a container for the block diagram. The *System* component is composed of one or more *Block* and *Line* components. The *Block* component represents a functional component, which can be of types *Subsystem, Primitive, Reference* or *S-Function*. A *Subsystem* block groups some functional control blocks, being composed of a unique *Sys-*

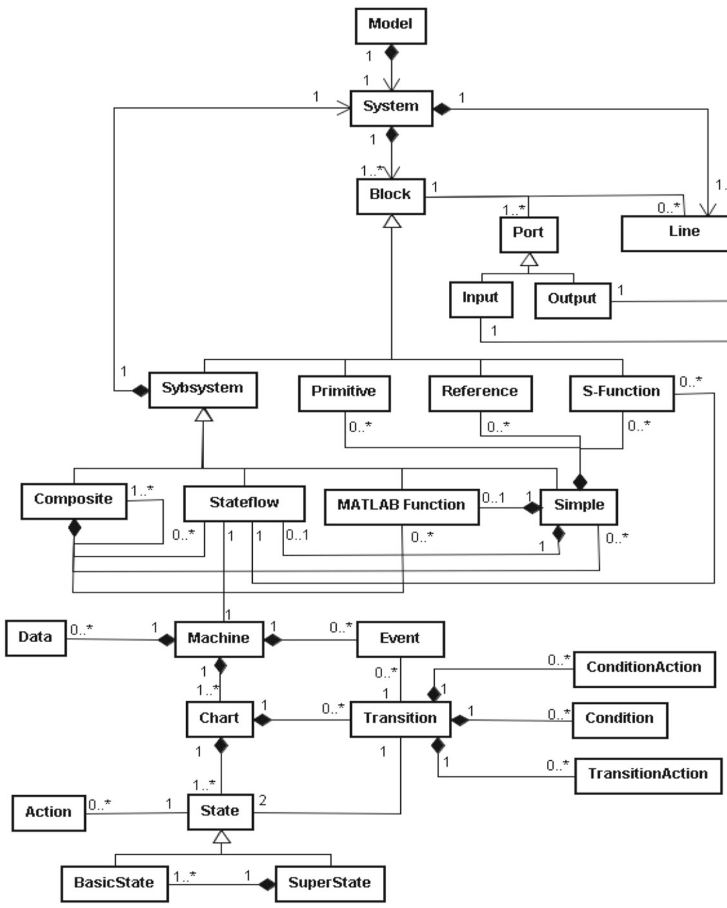

**Fig. 3** Simulink metamodel

*tem* component. This mechanism allows the hierarchical representation of a complex system. According to the Simulink metamodel, a *Subsystem* block can be of the following types: *Composite*, *Stateflow*, *MATLAB Function*, or *Simple*. A *Subsystem* block of the *Composite* type may be formed by one or several *Stateflow*, *MATLAB Function*, *Simple*, and/or *Composite* subsystem blocks. A *Stateflow* subsystem block stores the dynamic behavior of a system or component through a Stateflow diagram. A *Simple* subsystem block may be formed by any block types but not a block of the Composite type.

An important detail regarding the proposed Simulink metamodel is that the *Primitive* block represents all available pre-defined blocks in Simulink libraries. The *Reference* block represents a type of functional block created by the user and that can be reused in different Simulink models. The *S-Function* block represents those types of blocks that are capable of storing system functionalities written in programming languages such as Matlab, C, C++, or Fortran. Finally, the *Machine* component represents the hierarchy root component of the Stateflow diagram associated with the *Stateflow* subsystem block. There can be a maximum of one *Machine* component per Simulink model, formed by *Event, Data* and different *Chart* components. The *Chart* component represents the graphical view of the finite state machine,

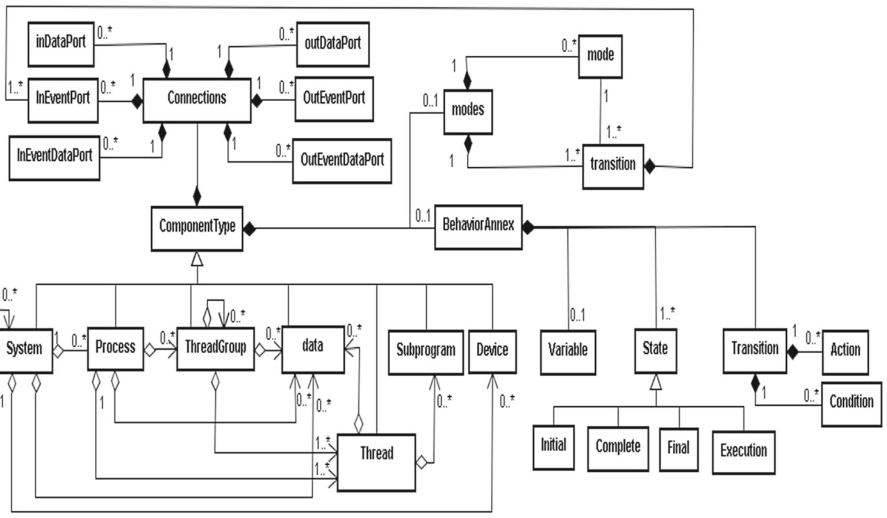

**Fig. 4** Simplified version of the AADL metamodel

and is composed of *State* and *Transition* components. The *State* component can be of the *BasicState* or the *SuperState* types, and can be associated to the *Action* and *Transition* components. A *Transition* component must be associated with a *State* component, and can be associated to the *Event* and *Condition* components.

Regarding the AADL metamodel, its official version published by SAE [17] is too complex for the needs of the present work. It has more than 100 entities and is subdivided into several ECORE files. As also noticed by other researchers, such complexity makes its use very difficult. Therefore, in order to ease the AADL metamodel analysis by AST, a simplified metamodel was created, as presented in Fig. 4. The part of the metamodel related with the model behavior (BehaviorAnnex component) was created using as reference the metamodel presented in [10], including additional behavioral elements extracted from the AADL Behavior Annex [18].

Observing the AADL metamodel in Fig. 4 one can say that the *ComponentType* represents the base for all AADL components: *System, Process, ThreadGroup, Thread, Data, Subprogram,* and *Device*. AADL models must necessarily include a *System* component, which must be composed of one or more *Process, Data,* and/or *Device* components - and even by other *System* components. *Process* represents a program organization with its own protected address space. It can contain *ThreadGroup*, *Thread*, and *Data* components. The *ThreadGroup* component represents a composition unit for the organization of the *Threads*, which represent a schedulable unit containing a concurrent function of the system. The *Data* component represents the data types and the static data of the source code. It can be composed of other *Data* components. *Subprogram* represents an executable code that can be invoked within a thread. *Device* represents components that make interface with the external environment, such as sensors, actuators, HMI, etc. They interact with the system by sending/receiving data and events. Like *Subprogram*, the *Device* component does not support any subcomponent.

The *ComponentType* represents all AADL components previously discussed. It can be associated with the *BehaviorAnnex* component, which constitutes the AADL behavioral annex. It provides an extension of the AADL language to allow a behavioral specification

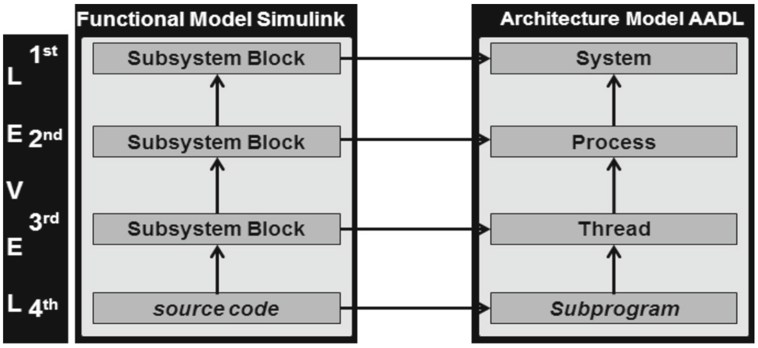

**Fig. 5** Correlation between Simulink and AADL models

to be attached to the components of an AADL model. The purpose of the behavioral annex is to allow the description of the component behavior through a state transition system with guards and actions [18]. Besides, the *ComponentType* can also be associated with modes of operation, that is, the designer can establish in which conditions the modes of operation of the system will be activated.

## 4.2 Correlation between Simulink and AADL metamodels

The functional and architectural models engineered during the development of a CPS can be considered complementary. As previously introduced, models from simulation tools can be used to characterize AADL components by providing their functionalities (see [6]). Such functionalities are modeled in AADL as *Subprograms*, which are considered as black-box artifacts typically associated with (invoked by) *Threads*. In fact, these artifacts are portions of source code implementing algorithms of interest to control the CPS.

The functions of a given system are often represented by block diagrams encapsulated in a Subsystem block. In a Simulink model, subsystem blocks normally group blocks that compose the control algorithm responsible for the operation of a particular function of the system. It was also observed that a subsystem block may be allocated hierarchically within another block subsystem, and so on. It is precisely this ability to hierarchical structuring that allows the identification of structural features in a Simulink functional model.

Figure 5 illustrates the correlation between the hierarchical structure of a Simulink functional model and the hierarchical structure of an AADL architectural model. In order to simplify the representation, the figure suggests a one-to-one relationship between Simulink subsystem blocks and AADL components. However, according to the metamodels presented in the previous section, a *composite* subsystem block (from Simulink) may contain of one or several subsystem blocks. Also, a *process* component (from AADL) can have one or more *threads* subcomponents.

The observed correlations only apply when considering hierarchical Simulink models. Obviously, if a flat Simulink model is created it is not possible to establish such correlations. Therefore, in order for designers to properly use AST, they should work with hierarchical Simulink models. In other words, this means that the Simulink blocks responsible for functions and/or sub-functions should be grouped into *Subsystem* blocks. Besides, it is very important that all *Subsystem* ports are properly defined (*signal name*, *port number* or *port number and signal name*), including the events and/or data (*int, double, boolean, etc.*) forwarded through them.

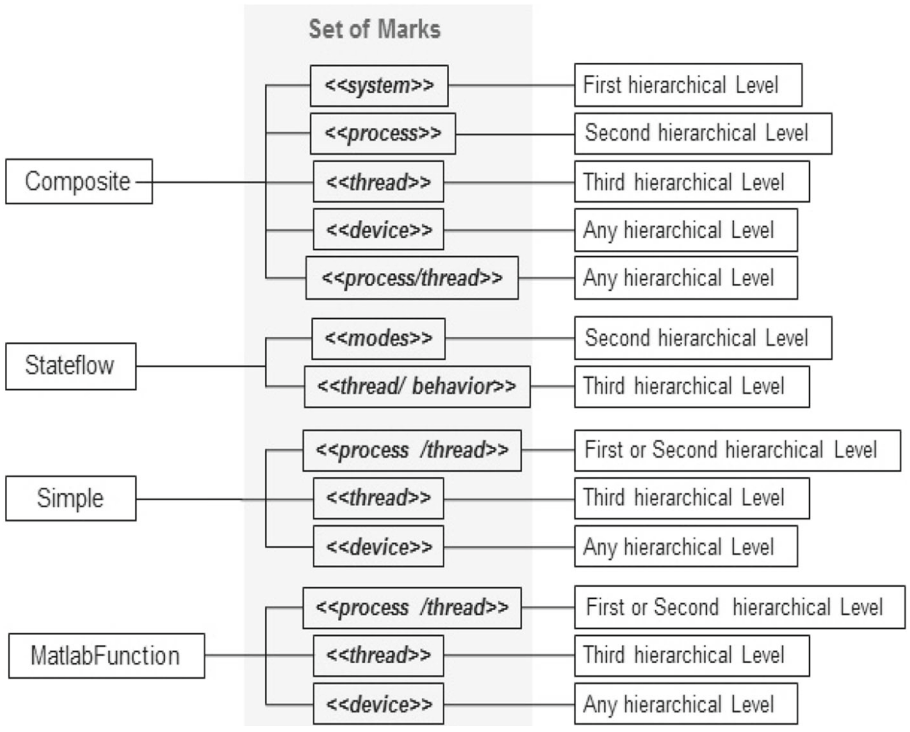

**Fig. 6** Set of marks that can be used to characterize the Simulink functional model

In AADL, devices represent external elements that interact with the system, like for example sensors, actuators, HMI, or the entire block representing the plant of the system under control. Such kind of elements normally exist in the Simulink model, but must be properly identified by the designer, as discussed in the next section.

The *Stateflow* Simulink block can be mapped to AADL either as modes of operation or as a thread behavior specification. It represents an operation mode when it is located in a higher hierarchical level of the Simulink model. It represents the thread behavior when in the lowest hierarchical levels.

## 4.3 Set of marks used in the transformation process

The *marks* represent AADL concepts that are used to characterize the *Subsystem* blocks in the functional model, and they indicate design decisions. Figure 6 presents the set of marks that can be used by the designer to mark (characterize) the elements of the Simulink functional model, so that it can go through a model transformation. The first column indicates the types of *Subystem* blocks that can be marked on Simulink functional models, the second column shows the marks that these blocks can receive, and the third column shows the hierarchical position that the *Subsystem* block should occupy within the Simulink functional model to receive the respective mark.

To understand the role of marks in the AST model transformation process, i.e., the AADL code generated from the “marked components”, the following situations are possible:

- When the *modes* mark is inserted into a *Subsystem* block of type *Stateflow* (normally positioned at the second hierarchical level of the Simulink model), a **modes section** is generated in the AADL component that contains (from the hierarchical perspective) the Stateflow block that received such mark.
- When the *process/thread* mark is inserted into a *Subsystem* block of type *Simple* (normally positioned in the first two hierarchical levels of the Simulink model), an AADL component of type *process* is generated, which includes a *thread* subcomponent that calls a *subprogram* subcomponent.
- When the *thread/behavior* mark is inserted into a *Subsystem* block of type *Stateflow* (normally positioned in the third level of the Simulink model), a *thread* component containing behavior specification is generated.

According to the proposed model transformation philosophy, a Simulink functional model can be marked either manually or automatically. To add the marks manually in a *Subsystem* block, the designer can use the feature of the Simulink tool to insert annotations and write the mark name there. The automatic marking of the model can be performed by the *AS2T plugin*, which was implemented in the scope of this work to automate the application of AST transformation rules.

When manually-marked, a Simulink functional model can be imported by the *AS2T plugin*, the markings are preserved by the transformation engine. When a non-marked model is imported by the *AS2T plugin*, the plugin automatically inserts suggested marks in *Subsystem* blocks according to their hierarchical position. The *device* mark is the only one that is not inserted by AS2T and, therefore, must be entered manually. While manual marking can potentially create different variations of the mapping for the same Simulink functional model, automatic marking always inserts the same marks in the same position for a given Simulink model.

## 5 The transformation engine

Due to mapping variations identified while structuring the proposed transformation engine, it was necessary to provide some kind of user interaction during the models transformation process. The user interaction can be performed by adding annotations in the Simulink model. Once the designer finishes annotating the Simulink model, the transformation engine can then generate the AADL source code.

AST deals with both the transformation of instances and the transformation of types. The transformation of types is straightforward, i.e., a certain element type from the source model is always mapped into the same way into the target model - the transformation is predefined. In the transformation of instances, an element from the source model can be mapped to different elements of the target model.

The user interacts with the transformation of instances by inserting annotations in the Simulink source model. Annotations are inserted into the model by using predefined names (*marks*) in the component annotation field. The most important mark is named *Device*, representing the external elements that interact with the system under design.

AST transformations rules are unidirectional, i.e., they are valid to transform a Simulink model into an AADL one—and not the other way around. Such transformation rules are classified into three categories: structural mapping, operation modes mapping, and behavioral mapping.

## 5.1 Structural mapping

The structural mapping constitutes the core of the proposed transformation approach, working with *Subsystem* blocks of types *Composite* and *Simple*. From such elements, the mapping generates the skeleton of an AADL specification. The set of transformations performed at this point are the following:

– *Composite-to-System* the first *Subsystem* block of type *Composite* is mapped to the *System* component in AADL.
– *Composite-to-Process* refining the *System*, the first *Subsystem* block of type *Composite* is mapped to the *Process* component in AADL when this block is further refined.
– *Composite-to-Process/Thread* when the block within the refined *System* is not further refined, it is mapped to an *Process* component in AADL that encapsulates a *Thread* that calls a *Subprogram* - so three AADL components are generated.
– *Composite-to-Device* a *Subsystem* block of type *Composite* with the *device* mark is mapped to the *Device* component in AADL.
– *Simple-to-Device* a *Subsystem* block of type *Simple* with the *device* mark is mapped to an AADL *Device* component.
– *Simple-to-Thread* if within a *Process*, a *Subsystem* block of type *Simple* is mapped to a *Thread* that calls a *Subprogram* - so two AADL components are generated.
– *Simple-to-Process/Thread* if not within a *Process*, a *Subsystem* block of type *Simple* is mapped to an *Process* component in AADL that encapsulates a *Thread* that calls a *Subprogram* - so three AADL components are generated here.

The structural mapping also defines that Simulink ports of types *signal name*, *port number*, and *port number/signal name* are respectively mapped to AADL as ports of the types *event port*, *data port*, and *event data port*. Data types forwarded through the ports of a *Subsystem* block are mapped as the data types in ports of types *data port* and *event data port* of the respective AADL component. By the end of the structural mapping, a data package with the data types that circulate through the ports should be generated. It should be highlighted that the connections of the *Subsystem* blocks from the Simulink model are preserved when mapped to AADL, which is done by means of adding connections to the generated AADL components.

## 5.2 Operation modes mapping

As the name suggests, the idea here is to obtain information about possible operation modes of the Simulink functional model and map them to the AADL architectural model. Therefore it is necessary to use information from the *Subsystem* blocks of type *Stateflow*, which can express state machines - named in Simulink as Stateflow Diagram. In AADL, the operation modes are associated with components and are represented by a modal state machine, according to the following transformations:

– *Stateflow-to-Operation Mode* a *Subsystem* block of type *Stateflow* in the same hierarchical level of the *Process* affects its father block (the block on the immediate higher hierarchical level). The corresponding AADL model (generated according to the structural mapping) receives *modes of operation*. The Stateflow transitions represent the transitions among modes and its actions represent the generation of the events that trigger the activation of the related AADL processes.

### 5.3 Behavioral mapping

Behavioral mapping is performed to extract behavioral information from the Simulink functional model and to map it to the AADL model. Similarly to the mapping of operation modes, it also manipulates *Stateflow* blocks. However here, for a matter of organization, in the Simulink model the Stateflow must be positioned within a block that can generate an AADL *Thread*.

In AADL, the specification of the internal behavior of a component is represented by a kind of state machine with guarded conditions and actions, which is properly described in the AADL Behavior Annex [18]. It should be part of the code section *annex behavior_specification* located within the thread, consisting of states and transitions. This mapping should comply with the following transformation:

– *Stateflow-to-behavior* a *Subsystem* block of type *Stateflow* representing the behavior that directly affects the *Thread* component further related with its containing block (the block on the immediate superior hierarchical level). Such *Thread* incorporates a code section called *annex behavior_specification*, which corresponds to a kind of state machine specification. It includes states, transitions, guards, and actions. The actions represent user-defined operations that are executed when the target state is reached.

### 5.4 Preliminary analysis

It is our claim that the AADL model generated by AST is preliminary, given that there is some additional work to complete the model. For instance, designers must manually specify how the generated model should be deployed in a proper execution platform (which must also be specified). After that, designers must decorate the AADL model (its components) with temporal properties, so that it can get ready for analysis and verifications.

Analyzing AST, one can say that it influences positively the automatic generation of the source code from the full system (functionalities and architecture), since it ensures that the models are consistent among them, establishing a clear correlation between the AADL model components and the respective Simulink model associated with it.

Regarding source code generation for the final application, currently the set of tools from OCARINA [11] can be used to generate source code in C or ADA given an AADL model as input.

## 6 Using AST within an UAV project

In this section we present a case study that shows how AST can be used during a project that aims at designing an Unmanned Aerial Vehicle (UAV). It is our understanding that such project represents many of the most relevant challenges related with the design of a complex CPS. Besides, given that the project was conducted within our research group, it was possible to have access to all project details and also to make use of the development method presented in Sect. 2. This allowed us to transform the Simulink model created for simulation purposes into an AADL model that represents the software architecture for the UAV system.

The UAV under consideration is detailed in [5,7]. It consists of a birotor with a tiltrotor configuration, i.e., an aircraft with two rotors that can be rotated individually by a dedicated servo. This allows the vehicle to perform vertical takeoff and landing (VTOL) like a helicopter, and also tilt the rotors horizontally to perform flights like a plane. The UAV under design

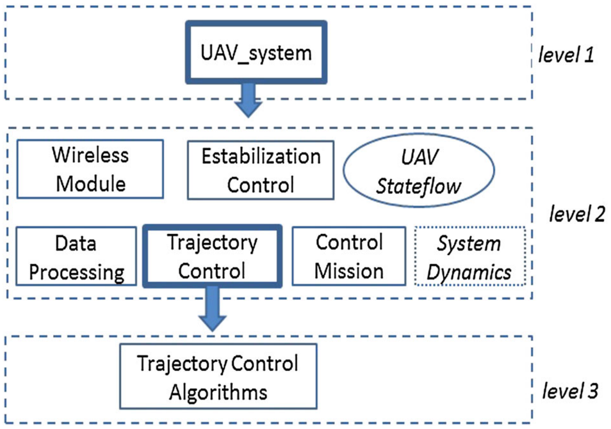

**Fig. 7** Hierarchical view of the UAV Simulink model

is autonomous, i.e., it can perform flights without pilot in accordance with a user-defined trajectory. A base station (BS) is used to specify the UAV mission (trajectory) and to perform data telemetry along the flight.

The Fig. 7 presents the hierarquical structure the UAV Simulink model used in this study as input for generating an AADL model that will represent the UAV embedded system architecture, as further discussed throughout this section.

## 6.1 Applying AST model transformation

To make use of AST, the first step is to check if the Simulink model (source model) is organized as a hierarchical structure. If this holds, the next step is to add the marks into the model, as described in Sect. 4.3. At least the *Device* mark should be added, as it is done those representing external devices or the physical model of the CPS under design. Once the Simulink model is marked, it is possible to perform the model transformation process.

A top-down presentation of the Simulink model is used to describe the model transformation process. The highest hierarchical level is shown in Fig. 8. It consists of three Subsystem blocks, one representing the UAV itself, another representing the Base Station (BS), and a third one representing the Remote Control (RC). Given that this case study focuses in the UAV and not in the BS nor the RC, only the former is marked as *system*, while the other two blocks are marked as *Devices*, as one can observe at the bottom of the respective blocks in Fig. 8.

To transform the UAV block we followed the **composite-to-system** mapping. The corresponding AADL code is shown in Fig. 9. It is possible to observe that this model contains a root system component called *architecture_uav* which has two subcomponents of the *devices* type called *d_remotecontrol* and *d_basestation* (lines 3 and 4) and a *system* component called *s_uavsystem* (line 5). Besides the subcomponents, the root system also has all connections, which were suppressed from Fig. 9 for a matter of simplification.

The next step is the decomposition of the UAV block (second hierarchical level), which is depicted in Fig. 10. Again, the selected mark for each block is presented below the block name. Those blocks with marks *process* or *process/thread* should be further detailed. The AADL code that represents this level is shown in Fig. 11. The five Simulink blocks of interest are mapped into subcomponents of the system component called *s_uavsystem* (lines 2 to 14).

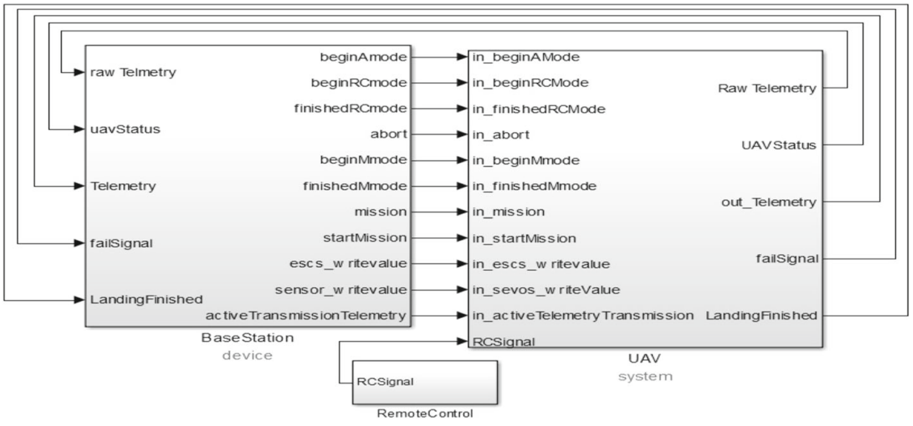

**Fig. 8** Simulink model of the 1st (highest) hierarchical level

```
1 SYSTEM IMPLEMENTATION architecture_uav.impl
2 subcomponents
3    d_remotecontrol: device d_remotecontrol.impl;
4    d_basestation : device d_basestation.impl;
5    s_uavsystem: system s_uavsystem.impl;
6 connections
...All connections are declared here...(lines 7 to 24)
25 END architecture_uav.impl;
```

**Fig. 9** AADL code for the first Simulink subsystem block

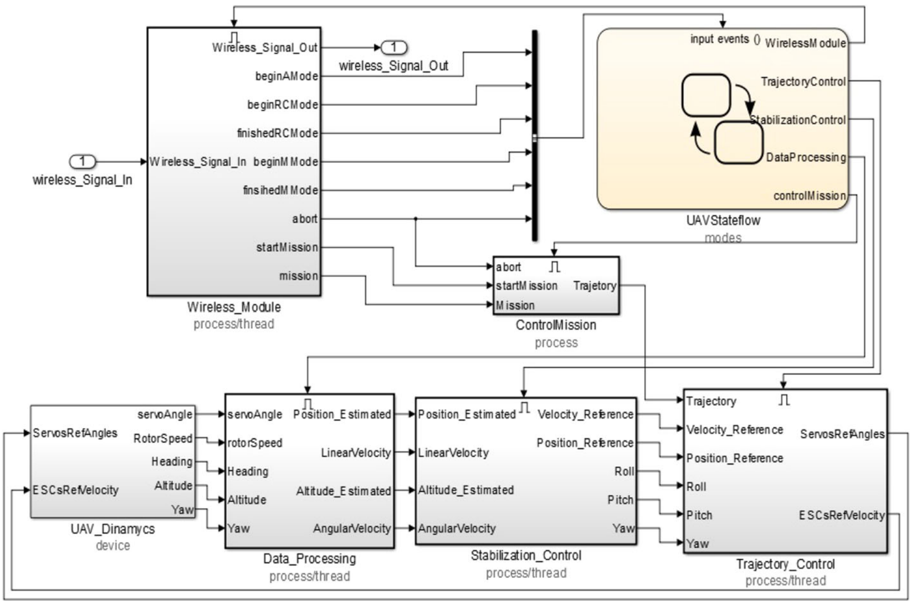

**Fig. 10** Simulink model (2nd hierarchical level): Refinement of block *UAV*

```
1 SYSTEM IMPLEMENTATION uav_system.impl
2 SUBCOMPONENTS
3   p_wirelessmodule : PROCESS p_wirelessmodule.impl in modes
4   (m_neutral, m_automatic_mode, m_radiocontroled_mode, m_maintenance_mode);
5   p_dataprocessing : PROCESS p_dataprocessing.impl in modes;
6   (m_automatic_mode, m_radiocontroled_mode, m_maintenance_mode,
7    m_emergencylanding_mode, m_returntohome_mode);
8   d_uavdinamycs : DEVICE d_uavdinamycs.impl;
9   p_stabilizationcontrol : PROCESS p_stabilizationcontrol.impl in modes
10  (m_automatic_mode, m_emergencylanding_mode, m_returntohome_mode);
11  p_trajectorycontrol : PROCESS p_trajectorycontrol.impl in modes
12  (m_automatic_mode, m_emergencylanding_mode, m_returntohome_mode);
13  p_controlmission : PROCESS p_controlmission.impl in modes
14  (m_automatic_mode, m_emergencylanding_mode, m_returntohome_mode);
15
16 CONNECTIONS
17   c1 : PORT p_trajectorycontrol.servosrefangles -> d_uavdinamycs.servorefangles;
18   c2 : PORT p_trajectorycontrol.escsrefvelocity -> d_uavdinamycs.escsvelocity;

... All connections are declared here...

34 modes
35    m_neutral: initial mode;
36    m_automatic_mode: mode;
37    m_radiocontroled_mode: mode;
38    m_maintenance_mode: mode;
39    m_emergencylanding_mode: mode;
40    m_returtohome_mode: mode;
41
42    m_neutral -[p_wirelessmodule.beginamode]-> m_automatic_mode;
43    m_automatic_mode -[p_wirelessmodule.landingfinished]-> m_neutral;
44    m_automatic_mode -[p_wirelessmodule.beginrcmode,]-> m_radiocontroled_mode;
45    m_automatic_mode -[p_wirelessmodule.failsignal]->
46    m_radiocontroled_mode;
47    m_automatic_mode -[p_wirelessmodule.failsignal]->
48    m_emergencylanding_mode;
49    m_automatic_mode -[p_wirelessmodule.abort]-> m_returtohome_mode;
50    m_neutral -[p_wirelessmodule.beginrcmode]-> m_radiocontroled_mode;
51    m_radiocontroled_mode -[p_wirelessmodule.finishedrcmode]-> m_neutral;
52    m_radiocontroled_mode -[p_wirelessmodule.abort]-> m_returtohome_mode;
53    m_radiocontroled_mode -[p_wirelessmodule.failsignal]->
54    m_emergencylanding_mode;
55    m_neutral -[p_wirelessmodule.beginmmode]-> m_maintenance_mode;
56    m_maintenance_mode -[p_wirelessmodule.finishedmmode]-> m_neutral;
57    m_returtohome_mode -[p_wirelessmodule.failsignal]->
58    m_emergencylanding_mode;
59    m_returtohome_mode -[p_wirelessmodule.landingfinished]->
60    m_neutral;
61    m_emergencylanding_mode -[p_wirelessmodule.landingfinished]->
62    m_neutral;
63 END s_uavsystem.impl;
```

**Fig. 11** AADL corresponding to the refinement of block *UAV*

For simplification, connections are suppressed (lines 19 to 33). Lines 34 to 62 are devoted for the modes, as further discussed.

According to the transformation criteria presented in the previous section, the Stateflow block present in Fig. 10 (*UAVStateflow*) is considered to represent operation modes. Figure 12 details such block, which is formed by six basic states that represent the operation modes of the system: Neutral, Automatic_Mode, RadioControled_Mode, Maintenance_Mode, Emer-

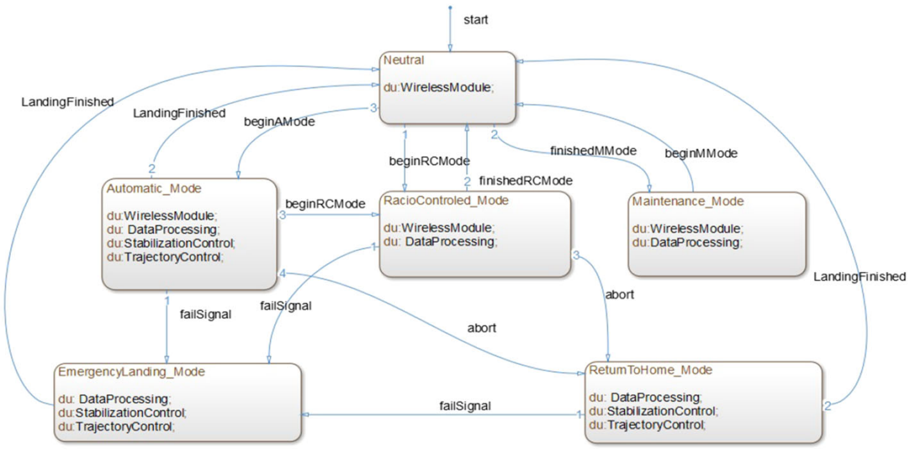

**Fig. 12** Stateflow diagram of the block *Mode Of Operation*

gencyLanding_Mode, and ReturnToHome_Mode. It also has a set of state transitions that are activated exclusively by events. The actions associated with the states are of type *During*, i.e., they are performed just after the new operation mode is triggered. Its transformation results in the AADL code also shown in Fig. 11—see lines 34 to 62, related to the *s_uavsystem component*. Lines 34 to 40 contain the operation modes and lines 42 to 62 represent the transitions among the operation modes. In AADL, the command *in modes* is used to specify the mode of operation of each process of the system component called *s_uavsystem*, and the operational mapping of AST extracts this information from the actions of the state of the Stateflow block that received the *modes* mark in the Simulink model.

Figure 13 shows the AADL code generated for the *Trajectory_Control* component. Since it is marked as *process/thread*, the code contains a process, a thread, and a subprogram call.

Finally we discuss the behavioral mapping. In order to show how the present work addresses the behavioral mapping, we changed the specification of the subsystem block identified as *ControlMission* in Fig. 10, providing to it a discrete behavior specification. Figure 14 illustrate such discrete behavior (stateflow diagram), which is positioned hierarchically within the *ControlMission* block – therefore it is in the third hierarchical level of the Simulink functional model. This diagram is intended to specify the expected behavior for controlling the execution of the mission received by the UAV. For transformation purposes, this stateflow block received the *thread/behavior* mark.

The transformation of the Stateflow in Fig. 14 results in the AADL code presented in Fig. 15. It has a thread with a code section named *annex behavior_specification*, containing the behavior specification.

## 6.2 Overview of the AADL model validation

In this section we present a few possible model analysis that can be performed with AADL models in general, such as response time and schedulability analysis. Besides, it we also address particularities of the AADL model that is automatically generated by the proposed AST.

In order to be able to perform model analysis with AADL models, its software components must be decorated with temporal information (periods and deadlines) and must also be

```
PROCESS p_trajectorycontrol
FEATURES
 trajectory : IN EVENT DATA PORT Base_Types_Simulink::integer ;
 velocity_reference : IN EVENT DATA PORT Base_Types_Simulink::integer;
 position_reference : IN EVENT DATA PORT Base_Types_Simulink::integer;
 roll : IN EVENT DATA PORT Base_Types_Simulink::integer;
 pitch : IN EVENT DATA PORT Base_Types_Simulink::integer ;
 yaw : IN EVENT DATA PORT ;
 servosrefangles : OUT EVENT DATA PORT Base_Types_Simulink::integer;
 escsrefvelocity : OUT EVENT DATA PORT Base_Types_Simulink::integer ;
END p_trajectorycontrol;

PROCESS IMPLEMENTATION p_trajectorycontrol.impl
SUBCOMPONENTS
  t_trajectorycontrol : THREAD t_trajectorycontrol.impl;
CONNECTIONS
  c1 : PORT trajectory -> t_trajectorycontrol.trajectory;
  c2 : PORT velocity_reference -> t_trajectorycontrol.velocity_reference;
  c3 : PORT position_reference -> t_trajectorycontrol.position_reference;
  c4 : PORT roll -> t_trajectorycontrol.roll;
  c5 : PORT pitch -> t_trajectorycontrol.pitch;
  c6 : PORT yaw -> t_trajectorycontrol.yaw;
  c7 : PORT t_trajectorycontrol.servosrefangles -> servosrefangles;
 c8 : PORT t_trajectorycontrol.escsrefvelocity -> escsrefvelocity;
END p_trajectorycontrol.impl;

THREAD t_trajectorycontrol
FEATURES
  trajectory : IN EVENT DATA PORT Base_Types_Simulink::integer;
  velocity_reference : IN EVENT DATA PORT Base_Types_Simulink::integer;
  position_reference : IN EVENT DATA PORT Base_Types_Simulink::integer;
  roll : IN EVENT DATA PORT Base_Types_Simulink::integer ;
  pitch : IN EVENT DATA PORT Base_Types_Simulink::integer;
  yaw : IN EVENT DATA PORT Base_Types_Simulink::integer;
  servosrefangles : OUT EVENT DATA PORT Base_Types_Simulink::integer;
  escsrefvelocity : OUT EVENT DATA PORT Base_Types_Simulink::integer;
END t_trajectorycontrol;

THREAD IMPLEMENTATION t_trajectorycontrol.impl
calls
   Mycalls: {
     P_Spg : subprogram programs_simulink::trajectorycontrol;
   };
END t_trajectorycontrol.impl;
```

**Fig. 13** AADL code for the *TrajectoryControl* block

properly bound to a target processor. The processors' frequency and their related scheduling protocol should also be defined after the proposed model transformation takes place. In the UAV study, its target platform (hardware) consists of two processors connected by a serial bus. Each processor has its own memory and both use the same scheduling protocol.

The AADL model of the UAV generated by AST was also subject to response time analysis. That is, we have analyzed the time required for an input signal to travel from an interface to the control system and for this to return a response. To perform this analysis, it was necessary

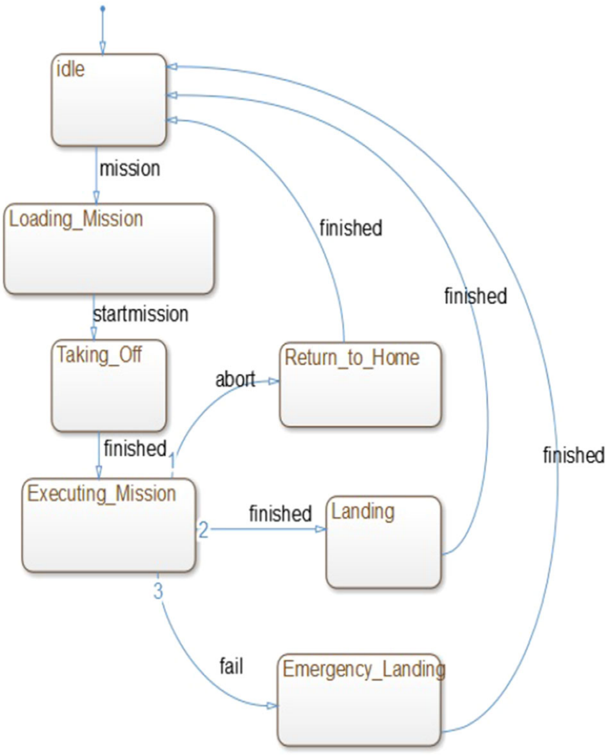

**Fig. 14** Diagram Stateflow of the ControlMission block

```
THREAD IMPLEMENTATION t_control_mission.impl
annex behavior_specification{**
states
  s_idle: initial complete state;
  s_loading_mission: complete state;
  s_taking_off: complete state;
  s_execution_mission:complete state;
  s_return_to_home: complete state;
  s_landing:  complete state;
  s_emergency_landing: complete state;
 transitions
    s_idle -[mission]-> s_loading_mission;
    s_loading_mission -[startmission]-> s_taking_off;
    s_taking_off[on dispatch -[finished]-> s_execution_mission;
    s_execution_mission -[finished ]-> s_landing;
    s_execution_mission -[abort]-> s_return_to_home;
    s_execution_mission -[fail]-> s_emergency_landing;
    s_landing -[finished]-> s_idle;        ]
    s_return_to_home -[finished]-> s_idle;
    s_emergency_landing -[finished]-> s_idle;
 **};
 END t_control_mission.impl;
```

**Fig. 15** AADL Model Level 3—thread *ControlMission*

to add flow specifications in the individual components of the system (*processes and threads*) and to specify an *end-to-end* flow in the system root of the AADL model. With the execution of the schedulability analysis of the AADL model of the UAV, it was possible to analyze the percentage of use of both processors in each operation mode of the system.

AADL models can also be subject to model checking whenever the Topcased tool chain presented in [3] is used. In the AADL model of the UAV, for instance, three simple checks were performed to verify the behavior of the *ControlMission* thread. The first check verified the absence of deadlocks and the second one verified the absence of temporal divergence. Finally, the third check verified if any state could cause the thread to block. The properties to be verified need to be described apart from the AADL model, using the LTL temporal logic.

### 6.3 Final discussion

The development of this study showed that by using AST one can create an AADL model to represent the architecture of a CPS using as input a Simulink functional model. Either in the performed study or in additional tests, most of the times the resulting AADL models were considered syntactically correct and semantically consistent by the OSATE AADL editor, which is the most widely used AADL editor. The few cases where this did not hold come from using flat Simulink models (with only one hierarchical level).

Assessing the readability of the the AADL models is also important, since AADL is primarily a textual language to be read by humans. Therefore we have compared the code generated by AST with the code generated by ADELE tool from a graphical AADL model that was equivalent to the one generated by AST. It was observed that the AADL code generated by AST is both more complete and readable. It is more complete because the respective AADL model can be instantiated directly since it has a root system, which does not happen with the textual AADL model generated by ADELE. Additionally, differently from the code generated by ADELE, the AST code for *event data port* and *data port* already includes data-type specifications. AST transformation engine generates a package with the data types transmitted through the ports of the Subsystem blocks. Furthermore, ADELE adds several fuzzy characters (mostly numbers) after the name of each component from the generated textual AADL model, which makes it difficult to be understood and maintained. However, we must recognize that even though the readability of the AADL model generated by ADELE is quite low, the fact that it has a graphical counterpart is a nice feature. Besides, the resulting AADL model could be transformed by AST, if they are applied manually.

As discussed in the previous section, AADL models can be formally verified, which contributes to increasing the correctness of the model under design. However, additional information should be added into the AADL components generated by AST, like timing and deployment information. In our point of view, this cannot be considered a limitation of AST, given that such information is not available in the functional model. Besides, adding such information is a common task to be performed during the detailed design.

## 7 Conclusions

The present paper addresses the problem of how to properly approach the design of the architectural model of a CPS. Differently from the most common design practice, where the architectural model is designed separately and then linked with the functional model (in the so-called deployment phase), the proposed approach suggests that the architectural model could be automatically generated from the functional model. This kind of approach avoids creating possible decoupled functional and architectural models and also promotes a truly "model-driven" design approach.

Given that functional and architectural models aim to represent different views of the system, the proposed mapping between both models can only be performed with some kind of user intervention. Therefore the proposed approach is designated as *Assisted Transformation of Models* (AST). It was created to transform Simulink models developed for simulation purposes into AADL models representing the system architecture. Essentially, AST consists of metamodels for the source and target languages, a set of marks (similar to stereotypes), and mapping rules. It also provides a tool support to automate the proposed model transformation.

AST promotes advances when compared to related works, as it covers the mapping of both structural and behavioral constructions. It is possible to argue that AST speeds up the generation of the architectural model because it avoids possible design mistakes in the definition of connections between ports of the software components and ensures data consistency between the ports involved in these connections. Additionally, due to the fact that AADL is being used a wider range of tests and checks are available when compared with the types analysis and verification mechanisms available for Simulink. As limitations of the overall approach, one can say that: (i) as AADL was essentially designed to represent structure, it is not possible to make use of the complete expressiveness power of the Stateflow diagrams; (ii) the designer still needs to specify additional information in the AADL model generated by AST so that it can be completed and submitted to specific analysis.

Using the TopCased verification chain, it was possible to perform successive transformations in the AADL model and to generate an equivalent automaton model that could be verified. In the future, AST could extend the existing transformation chain and thereby facilitate the integration of Simulink models with the TopCased environment. This should be better explored in the next steps of this work.

**Acknowledgments** Authors would like to thank the Brazilian funding agency CAPES for their support for the development of this work.

## References

1. (AVSI), A.V.S.I. (2010) The system architecture virtual integration program. http://savi.avsi.aero
2. Chkouri M, Bozga M (2009) Prototyping of distributed embedded systems using AADL. In: Baelen SV, Weigert T, Ober I, Espinoza H (eds) 2nd international workshop on model based architecting and construction of embedded systems (ACES-MB 2009), CEUR workshop proceedings, vol 507, pp 65–79
3. Correa T, Becker LB, Farines JM, Bodeveix JP, Filali M, Vernadat F (2010) A model-based design methodology for cyber-physical systems. In: 6th embedded real time software and systems conference (ETRS$^2$ 2010)
4. Delange J, Pautet L, Hugues J, De Niz D (2010) An MDE-based process for the design, implementation and validation of safety-critical systems. In: 15th IEEE international conference on engineering of complex computer systems (ICECCS 2010), pp 319–324. doi:10.1109/ICECCS.2010.12
5. Donadel R, Raffo G, Becker L (2014) Modeling and control of a tiltrotor UAV for path tracking. In: 19th IFAC World Congress, pp 3839–3844. IFAC. doi:10.3182/20140824-6-ZA-1003.01735
6. Feiler PH, Gluch DP (2012) Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language. Addison-Wesley, New York
7. Gonçalves F, Bodanese J, Donadel R, Raffo G, Normey-Rico J, Becker L (2013) Small scale UAV with birotor configuration. In: IEEE international conference on unmanned aircraft systems (ICUAS 2013), pp 761–768
8. Gonçalves F, Donadel R, Raffo G, Becker L (2013) Assessing the use of Simulink on the development process of an unmanned aerial vehicle. In: 3rd workshop on cyber-physical systems (CyPhy 2013)
9. INRIA ESPRESSO Team (2010) Polychrony. http://raweb.inria.fr/rapportsactivite/RA2010/espresso/uid27.html

10. Lasnier G, Pautet L, Hugues J, Wrage L (2011) An implementation of the Behavior Annex in the AADL-toolset Osate2. In: Perseil I, Breitman K, Sterritt R (eds) 16th IEEE international conference on engineering of complex computer systems (ICECCS 2011). IEEE Computer Society, pp 332–337. doi:10.1109/ICECCS.2011.39
11. Lasnier G, Zalila B, Pautet L, Hugues J (2009) Ocarina: an environment for AADL models analysis and automatic code generation for high integrity applications. In: 14th Ada-Europe international conference on reliable software technologies (Ada-Europe 2009). Springer, New York, pp 237–250. doi:10.1007/978-3-642-01924-1_17
12. Le Guernic P, Gautier T, Le Borgne M, Le Maire C (1991) Programming real-time applications with SIGNAL. Proc IEEE 79(9):1321–1336
13. Lee E (2008) Cyber physical systems: design challenges. In: 11th IEEE international symposium on object oriented real-time distributed computing (ISORC 2008), IEEE Computer Society, pp 363–369. doi:10.1109/ISORC.2008.25
14. Mathworks T (2011) Using Simulink. http://www.mathworks.com/access/helpdesk/help/pdf_doc/simulink/sl_using
15. Miller J, Mukerji J (2000) MDA Guide Version 1.0.1. Technical Report, Document omg/2003-06-01, Object Management Group
16. Raghav G, Gopalswamy S, Radhakrishnan K, Delange J, Hugues J (2009) Architecture driven generation of distributed embedded software from functional models. In: Ground vehicle systems engineering and technology symposium (GVSETS 2009)
17. SAE (2006) SAE AADL meta model and XML/XMI. http://www.aadl.info/aadl/currentsite/tool/metamod.html
18. SAE (2011) SAE Architecture Analysis and Design Language (AADL); Annex vol 2: Annex B: Data Modeling Annex, Annex D: Behavior Model Annex, and Annex F: ARINC653 Annex. http://standards.sae.org/as5506/2
19. Selic B, Gérard S (2014) Modeling and analysis of real-time and embedded systems with UML and MARTE. Morgan Kaufmann, Burlington
20. The CESAR Project (2010) Cost-eficient methods and processes for safety relevant embedded systems. http://www.cesarproject.eu
21. Yu H, Ma Y, Glouche Y, Talpin JP, Besnard L, Gautier T, Guernic PL, Toom A, Laurent O (2011) System-level co-simulation of integrated avionics using Polychrony. In: 2011 ACM Symposium on applied computing (SAC 2011). ACM, New York, pp. 354–359. doi:10.1145/1982185.1982263
22. Zowghi D, Coulin C (2005) Requirements elicitation: a survey of techniques, approaches, and tools. In: Aurum A, Wohlin C (eds) Engineering and managing software requirements. Springer, Berlin, pp. 19–46. doi:10.1007/3-540-28244-0_2