StateML$^+$: From Graphical State Machine Models to Thread-Safe Ada Code[†]

Diego Alonso[1], Cristina Vicente-Chicote[1], Juan A. Pastor[1], Bárbara Álvarez[1]

[1] Departamento de Tecnologías de la Información y las Comunicaciones
División de Sistemas y Ingeniería Electrónica (DSIE)
Universidad Politécnica de Cartagena, 30202 Cartagena, Spain
{diego.alonso, cristina.vicente, juanangel.pastor, balvarez}@upct.es

Abstract. This paper presents the StateML$^+$ tool aimed at designing state-machines and automatically generating thread-safe and multi-tasking modular Ada code from them, following a Model-Driven Engineering approach. The StateML$^+$ meta-model is an extension of a previous version, and now it offers improved modeling capabilities, which include regions and macro-state definition. In this paper, a case study regarding the design of a robotic system will be used to demonstrate the benefits of the proposed approach.

Keywords: Model-Driven Engineering, Model-To-Text Transformation, Finite State Machines, Thread-Safe Code Generation, Eclipse platform.

1 Introduction

Model-Driven Engineering (MDE) technologies offer a promising approach to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively [1]. Objects are replaced by models, and model transformations appear as a powerful mechanism to automatically and incrementally develop software [2].

The work presented in this paper starts from the definition of the *ACRoSeT* [3] abstract architectural framework, aimed at developing abstract software components for tele-operated robots. This framework allows designers to define the software architecture of a robotic system in terms of abstract (platform independent) robotic components. *ACRoSET* components are designed taking into account both structural and behavioral aspects.

Although the adoption of *ACRoSET* for component-based robotic system design has demonstrated many advantages, the translation of its abstract components into concrete ones has not been automated yet and, thus, it remains an error-prone process. This is one of the current aims of our research group and this paper covers it partially.

[†] This research has been funded by the Spanish CICYT project MEDWSA (TIN2006-15175-C05-02) and the Regional Government of Murcia Seneca Program (02998-PI-05).

To tackle the problem of automatically translating *ACRoSET* abstract components into concrete ones, we propose a MDE approach based on a previous experience, already published in [4]. In that work, we presented a basic state-machine meta-model, called StateML, and a graphical modeling tool built on top of it, which allowed designers to depict and to validate very simple state-machine models. These models could then be automatically translated into Ada code using a model-to-text transformation, also implemented as part of that work.

State machines provide very powerful behavioral descriptions. This is why they are quite commonly used for modeling general-purpose processes and, in particular, why they have been extensively adopted by the robotics community. Even when using a Component-Based Software Development (CBSD) [5] approach for robotic application design, as the one proposed in *ACRoSET*, state-machines are a very appropriate and natural way for describing component behavior, since they allow designers to define how components react to different external and internal stimuli. In addition, state machines provide a very natural and precise notation for describing aspects such as concurrency.

However, in order to model *ACRoSET* abstract components, the state-machine models built using the previous StateML tool presented in [4] was not expressive enough, since it did not include mechanisms to model concurrency. Thus, instead of tackling the whole problem of translating *ACRoSET* abstract components into concrete ones, we decided to first complete the state-machine models and the translation of the component behavior part, leaving the structural aspects for a later stage.

In this vein, this paper presents the extended StateML$^+$ meta-model, which includes all the concepts needed to model the behavior of *ACRoSET* abstract components, including those related to concurrency. Besides, the new tools implemented on top of this meta-model are also presented, i.e. a new graphical model editor and a new automatic model-to-code transformation, which generates a thread-safe Ada code implementation of the input state-machine model. Although developed in the context of *ACRoSeT*, StateML$^+$ can be used as a stand-alone tool by any designer who wants to generate a multi-threaded Ada skeleton from a hierarchical state machine.

Before entering into details, the following section presents an outline of the research goals covered in this paper. Then, the rest of the paper is organized as follows. Firstly, section 2 presents the extended StateML$^+$ meta-model, and the graphical modeling tool implemented to support the newly added elements. This section also presents a case study on robotics that will be used through the rest of the paper to illustrate the benefits of the proposed approach. Then, the automatic model-to-code transformation, from StateML$^+$ models to thread-safe Ada code, is presented in section 3. Finally, section 4 presents the conclusions and some future research lines.

1.1 Goals of the Paper

In reactive systems, software commonly interacts simultaneously with multiple external elements (sensors, actuators, robots, conveyor belts, etc.). Actually, the real-world is inherently concurrent and this must be somehow captured in software

applications. This obviously requires using platforms and programming languages which provide concurrency mechanisms.

As previously stated, the StateML meta-model did not offer any mechanism to model concurrency, although in [4] we proposed this extension as a future work. The main goal of this paper is to show how we have addressed this extension presenting the improved StateML$^+$ meta-model, which now can deal with concurrency aspects. To achieve this goal, the following sub-goals have been addressed:

- Firstly, the state-machine meta-model was extended with new concepts in order to improve its modeling capabilities. Among others, it now includes orthogonal regions to represent independent and concurrently active states. The extension of the meta-model implied the addition of a comprehensive set of OCL (*Object Constraint Language*) [6] expressions in order to complete the syntax and the semantics of the meta-model, as it will be further explained in section 2.1.
- A new graphical modeling tool was also developed to allow designers to graphically define state-machine models and to validate them against the meta-model and the set of additional OCL constraints. This tool was validated building different robotic-related case studies, such as the one presented in section 2.2.
- A suitable design pattern [7] had to be selected in order to perform the model-to-code transformation. This implied reviewing some of the architectural patterns that could cope with the *run-to-completion* semantics associated to the state-machine artifacts. After a careful reviewing process, the *Reactor Pattern* [8] was finally selected, as it will be further justified in section 3.1.
- Finally, a new *MOFScript* [9] model-to-code transformation was implemented in order to generate thread-safe Ada code from any input state-machine model. This transformation, which implements the selected *Reactor Pattern,* is detailed in section 3.2.

After covering these goals, the paper will present some conclusions and future research lines.

2 StateML$^+$: Improving FSM Modeling Capabilities

As stated in the introduction, this paper presents StateML$^+$, which is an improved version of StateML, already presented in [4]. The state-machine meta-model included in StateML was designed as a quite simplified version of the UML 2.x [10] counterpart.

In StateML, designers could model state-machines consisting of states linked by transitions, which could be external or internal, depending on whether the state was actually exited or not. Designers could also include in their models one initial pseudo-state (to initialize the state-machine and to mark the first state to be executed) and one or more final states (to mark the state-machine execution end). The StateML meta-model was enriched with a complete set of OCL constraints in order to assert that models built from it were syntactically and semantically correct.

In spite of the good results obtained by the StateML tools, it was quite clear that its modeling capabilities were very limited, particularly to modeling real world system

behavior. As already stated, one of the biggest limitations of StateML was the lack of macro-states and orthogonal regions, since (1) macro-states help avoiding state explosion in state-machines [11], and (2) orthogonal regions make it possible to model the concurrent aspects of a state.

The new StateML$^+$, presented in this paper, tries to overcome the limitations of the previous version, extending the underlying state-machine meta-model with orthogonal regions (among other modeling elements), and, thus, providing extended modeling capabilities. Accordingly, the graphical modeling tool (see section 2.2), and the automatic model-to-Ada transformation (see section 3) have also been extended to support the new StateML$^+$ extended meta-model.

2.1 The StateML$^+$ Extended Meta-Model

The StateML$^+$ extended meta-model is shown in Fig. 1. As justified before, when compared to the previous version (StateML meta-model [4]), the main difference is the inclusion of the `Region` concept, which now plays a central role. The elements included in the meta-model and the relationships existing between them are briefly described next.

- **`Region`**. As previously stated, this concept has been newly added to the meta-model. Each `Region` is contained in a `State`, but a special one, called *`topRegion`*, which is contained in the `StateMachine` itself. In this new version, `Regions` contain `Vertexes` and `Transitions`, which were directly stored in the `StateMachine` in the previous StateML version.
- **`StateMachine`**. As already explained, in this new version of the meta-model, the `StateMachine` contains a *`topRegion`* instead of directly containing the `Vertexes` and `Transitions` that constitute the state machine. Besides, it has a `unexpectedTransitionPolicy` property which can take values {`IGNORE`, `NOTIFY_ERROR`}. According to the value of this property, the `StateMachine` will react differently when it receives an event which does not trigger any of the outgoing transitions of the current state. Specifically, the `StateMachine` will ignore the unexpected event if the `IGNORE` value has been selected, and it will call an error handler otherwise (`NOTIFY_ERROR` value selected). This fact is considered as a "semantic variation point" in the UML 2.x specification and was outlined in [4] as a future improvement of StateML.
- **`State`**. In this extended version of the meta-model it is possible to create `States` containing `Regions` which may contain other `States` (and `Transitions`), up to any nesting level. Thus, this structure allows for creating macro-states. The `State` element has also been enriched with three boolean properties named `hasOnEntry`, `hasDo`, and `hasOnExit`, which allow designers to establish whether the implementation of the `State` will have any of these operations. These boolean properties are parsed during the model-to-Ada transformation step, as it will be widely explained in section 3.2.

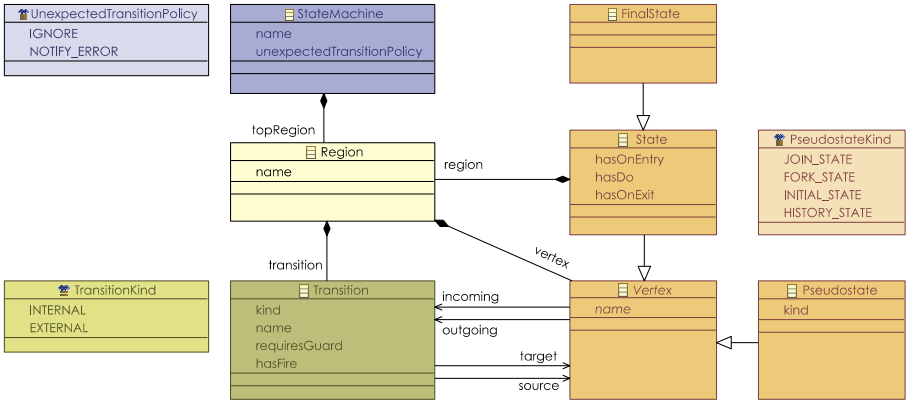

Fig. 1. The StateML^{+} extended meta-model.

- **`Transition`**. This element remains similar to the one included in the previous version. It keeps the `kind` property, which can take values {`INTERNAL`, `EXTERNAL`} as before, and includes two new boolean properties `requiresGuard` and `hasFire`, which control whether the designer wants a `Transition` to have a guard and a fire operation. These boolean properties, together with those added to the `State` element, are used during the model-to-Ada transformation step.
- **`Pseudostate`**. This element remains identical to the one included in StateML, i.e. it only includes a property `kind` of type `PseudostateKind`. However, as shown in Fig. 1, the `PseudostateKind` enumerated type has been enriched with new elements to cope with the new needs derived from the inclusion of `Regions` in the meta-model. Thus, the property `kind` can now take values {`INITIAL_STATE`, `HISTORY_STATE`, `JOIN_STATE`, `FORK_STATE`}. The syntax, and the semantics of these pseudo-states has been taken form the one defined in the UML 2.x specification [10], and it is summarized next.
  - » **`INITIAL_STATE`** / **`HISTORY_STATE`**. Initial vertexes represent a default vertex that is the source for a single `Transition` to the default `State` of a `Region`. In the case of `HISTORY_STATE`s, they have the ability to "remember" the last active `State` the `Region` was in before exiting (the `Region`). Thus, when the `Region` is entered again, the `HISTORY_STATE` re-starts the last active `State` execution again. There can be at most one initial vertex (either `INITIAL_STATE`s or `HISTORY_STATE`s) in a `Region`. The outgoing `Transition` from the initial vertex can not have either a fire operation or a guard.
  - » **`JOIN_STATE`**. `JOIN_STATE Pseudostates` allow to merge several `Transitions` emanating from source `States` belonging to orthogonal `Regions`, enabling their synchronization. The `Transitions` entering a `JOIN_STATE Pseudostate` cannot have either guards or fire operations.
  - » **`FORK_STATE`**. `FORK_STATE Pseudostates` allow to split an incoming `Transition` into two or more `Transitions` terminating on orthogonal

target `States` (i.e., `States` in different `Regions` of a macro-state). The `Transitions` outgoing from a `FORK_STATE Pseudostate` must not have guards or fire operations.

The StateML$^+$ meta-model extension made arise many additional syntactic and semantic constraints, which could not be directly expressed in the meta-model, given the limitations of using a MOF [12] meta-class diagram. As a consequence, a comprehensive set of OCL [6] constraints had to be implemented to cope with the new modeling restrictions. For space reasons, only two of these constraints are included in this paper (see Table 1).

Next section presents the new graphical model editor built to support the new modeling capabilities of StateML$^+$, together with a robotic system case study developed using this new tool.

Table 1. Two of the OCL constraints included to complete the formal syntax of StateML$^+$

Target domain element: `Transition`

Description: *Pseudostates cannot have internal transitions*

OCL rule:

```
self.kind=TransitionKind::INTERNAL
   implies not(self.source.oclIsTypeOf(Pseudostate))
```

Target domain element: `Transition`

Description: *External Transitions from `INITIAL`, `HISTORY`, and `FORK` and to `JOIN` Pseudostates must have `requiresGuard`=false and `hasFire` =false*

OCL rule:

```
    ((self.kind=TransitionKind::EXTERNAL)    and

   ((self.source.oclIsTypeOf(Pseudostate))
or
    (self.target.oclIsTypeOf(Pseudostate))) and

   ((self.source.oclAsType(Pseudostate).kind=
          PseudostateKind::INITIAL_STATE )
or (self.source.oclAsType(Pseudostate).kind=
          PseudostateKind::HISTORY_STATE )
or (self.source.oclAsType(Pseudostate).kind=
          PseudostateKind::FORK_STATE )
or (self.target.oclAsType(Pseudostate).kind=
          PseudostateKind::JOIN_STATE )))
implies
    (self.requiresGuard=false and self.hasFire=false)
```

2.2 Building Graphical StateML$^+$ Models: A Case Study on Robotics

A new graphical modeling tool has been developed to support the new modeling capabilities of StateML$^+$ and also the new restrictions needed to complete the meta-model specification, as previously introduced in section 2.1.

As the previous version of the tool, the StateML+ graphical model editor was implemented using the Eclipse Graphical Modeling Framework (GMF). Thus, we followed a similar approach as the one described in [4] but, in this case, the new elements included in the StateML+ meta-model and the new OCL constraints were taken into account.

As shown in Fig. 2, the new tool offers a richer palette, where the user can now select new elements such as `Region` or the new different kinds of pseudo-states. Besides, users can validate their models both against the meta-model and against the newly added set of additional OCL constraints, thus assuring that their models are totally correct before proceeding to the model-to-code transformation step.

In order to test both the new modeling capabilities of the StateML+ meta-model, the graphical modeling tool built on top of it, and the automatic model-to-Ada code transformation implemented afterwards, we needed a simple yet real-world case study. Given that, as stated in the introduction, our application domain is very related to robotics, we decided to use the state machine model proposed in [13] and depicted in Fig. 2.

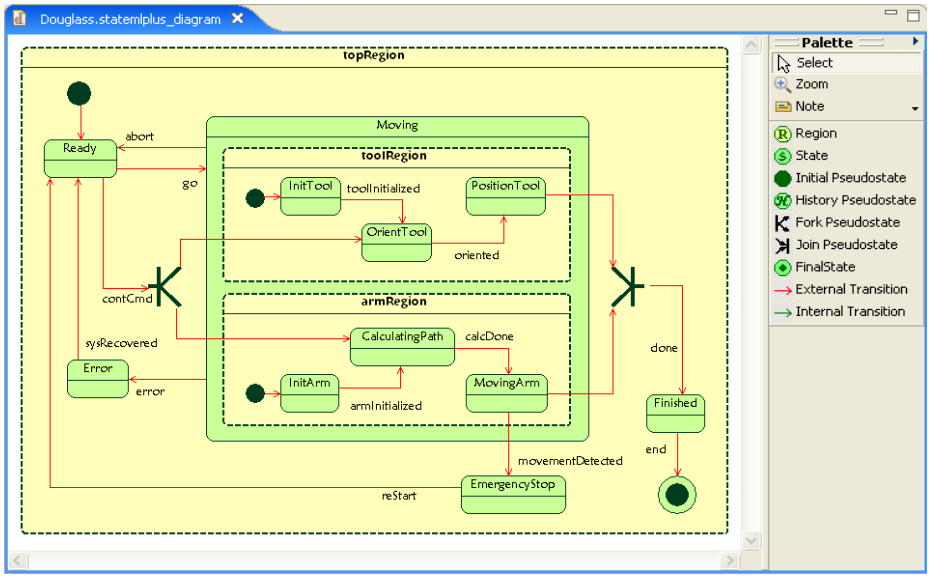

Fig. 2. State-machine model depicted using the StateML+ graphical modeling tool. This state machine models the behavior of the robotic arm used as the case study in this paper.

The state-machine model shown in Fig. 2 models the behavior of a robotic arm which holds a tool (e.g. a gripper, a welder, etc.). As the movement of the arm and the movement of the tool are independent of each other, they have been modeled with two orthogonal regions (`toolRegion` and `armRegion`) to show this independence. Each of these regions contains the states in which the tool or the arm can be in, independently of the current state of the other region. The rest of the states

(i.e. `Ready`, `Error`, `EmergencyStop` and `Finished`) are directly contained in the `topRegion` of the state-machine. In addition:

- An `initial` pseudo-state has been added to each region to mark its default initial state, i.e. where the region starts its execution. This restriction is checked by an OCL expression.
- A `fork` pseudo-state has been included in the `topRegion` to split the `contCmd` transition from the `Ready` state to two states included in the orthogonal regions defined in the `Moving` state.
- A `join` pseudo-state has also been added to the `topRegion` to synchronize the outgoing transition from the `Moving` state, i.e. from two of the internal states belonging to its internal regions.

This case study uses all the new elements added to StateML$^+$, showing a simple yet rich enough case study. This case study has served also as the input to test our model-to-Ada code transformation that is explained in the following section.

3 From StateML$^+$ Models to Thread-Safe Ada Code

The meta-model of a system plays a central and fundamental role in the MDE paradigm, since it is the basis that supports the rest of the MDE development process, namely: model creation and model transformations [14]. Therefore, changing the meta-model implies updating the graphical model editor and the model-to-Ada transformation that were previously developed for the StateML tool.

The model-to-Ada transformation, which is described in section 3.2, has suffered a deep modification and now generates Ada code that can cope not only with the new modeling capabilities of StateML$^+$ but that is also ready to be included in any multi-tasking application. Besides, we have seized the fact that we should update the model transformation to include some of the characteristics that were outlined as "future work" in the previous StateML tool [4].

Section 3.1 briefly describes some of the architectural design pattern that could have been used to implement the concurrency aspects derived as a consequence of the improvement of the StateML tool, together with the main characteristic of the chosen implementation pattern: the *Reactor/Dispatcher* pattern.

3.1 Decoupling State Activities Execution from State-Machine Management: Using the Reactor Pattern.

One of the main challenges of the model-to-code transformation of the state machine artifact resides in the *run-to-completion* semantics associated to the state machine. The run-to-completion semantics, as appears in the UML superstructure (see [10], chapter 13) specifies that:

> "*An event occurrence can only be taken from the pool and dispatched if the processing of the previous event occurrence is fully completed. Before commencing on a run-to-completion step, a state machine is in a stable state*

configuration, with all entry/exit/internal activities (but not necessarily state (do) activities) completed. The same conditions apply after the run-to-completion step is completed. Thus, an event occurrence will never be processed while the state machine is in some intermediate and inconsistent situation. The run-to-completion step is the passage between two state configurations of the state machine. The run-to-completion assumption simplifies the transition function of the state machine, since concurrency conflicts are avoided during the processing of event, allowing the state machine to safely complete its run-to-completion step".

This run-to-completion requirement was not a problem in the previous StateML tool, as it does not allow the creation of regions, and thus there is always one and only one active state. But, in StateML$^+$, the presence of regions breaks this rule, as there may be many active states inside any given macro-states. A possible alternative consists of the execution of every active state in its own thread. But using multi-threading in such an uncontrolled way has the following drawbacks:

- Threading may be inefficient and non-scalable.
- Threading may require the use of complex concurrency control schemes throughout the state machine code.
- Concurrency usually implies longer development times, as it has its own problems.

In our case, it was very advisable to come up with a scalable solution according to the number of states appearing in the model, and that could shorten implementation time from the former model-to-Ada transformation.

The *Reactor/Dispatcher* architectural pattern [8] provides a solution that accomplish this requirement. This pattern allows event-driven applications to de-multiplex and dispatch service requests that are delivered to an application from one or more clients. The handlers of these service requests are registered within a reactor thread that runs an infinite loop in which the registered handlers are run sequentially. The reactor thread provides a way to add and remove event handlers at run-time, so the application can adapt itself to changing requirements. The use of the Reactor pattern has allowed us to:

- Achieve "concurrency" for the states contained in orthogonal regions, eliminating the need of complex synchronization mechanisms and shortening in this way the implementation time of the transformations.
- Decouple the state machine management and the short duration activities involved in the run-to-completion semantics of the state-machine (transitions, entry and exit actions execution) from the long duration activities that may be associated to states.

However, the Reactor pattern is not suitable for long duration activities, as they are executed sequentially by the Reactor. Thus, the main liability has been the need of constraint the duration of the activities associated to states. Activities should be of short duration, though they are repeated every time the Reactor executes its cycle while the state machine remains in the given state. In this case, the reactor pattern can be seen as a cyclic executive scheduler. Long duration activities are more effectively handled using separate threads. This can be achieved via other patterns, such as

Active Object or Half Sync/Half Async [8]. In general, the Reactor pattern is difficult to test and debug, but in our case simplicity helps avoiding these drawbacks.

3.2 Model-To-Ada Transformation: Implementing the *Reactor Pattern*

As said before, the Reactor pattern will be use to embed the run-to-completion semantics of the state-machine artifact in the resulting Ada code implementation of a StateML$^+$ model. In this case, the handlers executed by the reactor task are the `Do_Activity` associated to each state, while the events that trigger their execution are the transitions of the state-machine. When a transition occurs, the activity associated to the new state is registered in the reactor while the activity associated to the old state is removed from it. The same happens when entering or exiting a macro-state with orthogonal regions.

Fig. 3 shows the UML package diagram that describes the structure of the Ada code generated after the model-to-code transformation, which generates:

- A main procedure, called **Simulator**, which contains a command-line program that can command and control the generated state machine. This program is used only for different testing purposes. The state-machine is completely usable on itself without this procedure.
- A package, which name depends on the name of the state-machine model name (**Main_Fsm** in Fig. 3)**,** implements all the structure and control logic of the state machine. This package contains: (1) the private child packages (depicted in Fig. 3 with a thicker border) that implement the different modules of the Reactor pattern as it will be explained in the following items, (2) the parameter-less procedures that signal the event occurrence that may trigger the fire of a transition of the state-machine, (3) a function to get the state machine current state and the corresponding types to correctly deal with it, and (4) the procedures that notify that an unexpected event has happened (these procedures are created when using the `NOTIFY_ERROR` value in the `unexpectedTransitionPolicy` property of the `State-machine`, see section 2.1).
- A private child package, named **Main_Fsm.Fsm_Task**, which contains a protected buffer to store event signaling and the task that controls the flow of states. The protected buffer accomplishes two main objectives: (1) it decouples event signaling from event processing, just as the Reactor pattern specifies, by using the *Command* design pattern [7], and (2) it makes the use of StateML$^+$ state-machines in multi-threaded applications possible. The task created in this package is in charge of

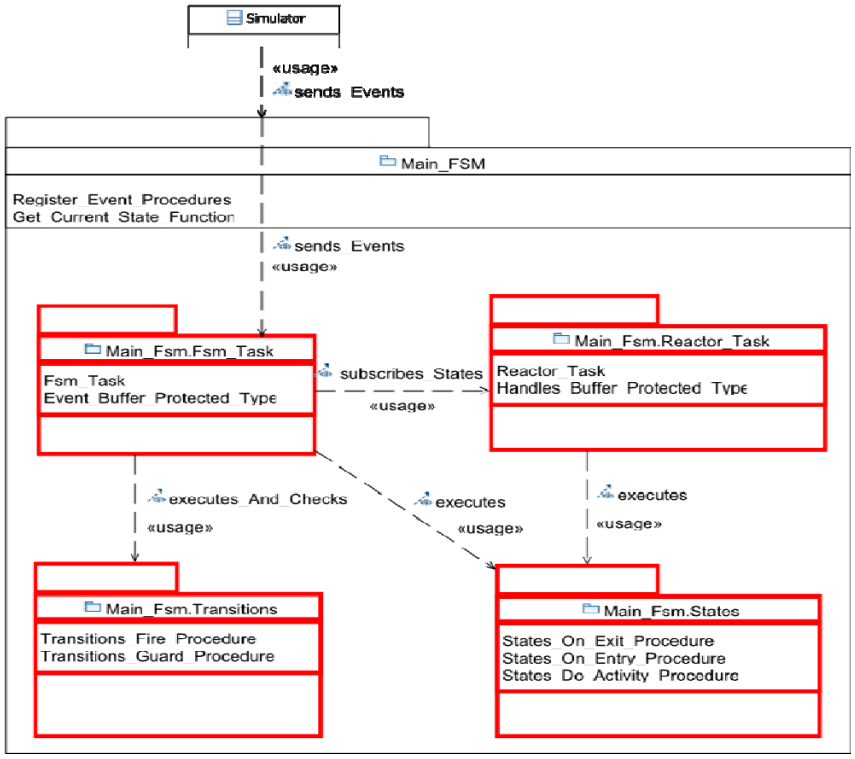

Fig. 3. UML package diagram showing the structure of the generated Ada code. Highlighted packages (those with a thicker border) are private childs of the outer package.

controlling the state-machine and changing its state, depending on the received event. This task embodies the *run-to-completion* semantics that is associated to the state-machine execution, which includes event processing, entry/exit/fire activities execution and state change, but not `do_activity` execution.

- Another private child package, named **Main_Fsm.Reactor_Task,** that plays the role of the reactor that corresponds to its name. This package contains the list of currently active states (those whose `Do_Activity` has to be executed), together with the reactor task in charge of sequentially executing them. The list of activities to be executed is maintained and updated by the **Main_Fsm.Fsm_Task** when it processes an event that changes state. This design decision frees the **Fsm_Task** from executing the `Do_Activity`, so it can process the next event as soon as it completes the *run-to-completion* step.
- The private child package **Main_Fsm.Transitions** contains the specification and empty bodies of (1) the procedures that describe the activity that should be executed when a transition is fired (named `Fire_XXX_Transition`), and (2) the functions that check whether the transition should be executed or not, that is, the guard of the transition (`Can_Fire_XXX_Transition`). All of these

subprograms are automatically generated if the corresponding attribute of the `Transition` is set (`hasFire` for generating the fire procedure and `requiresGuard` for the guard function). Of course, all these subprograms must be filled-in by the developer of the final application. Besides generating the specification of these subprograms, the transformation also generates the corresponding calls to these subprograms inside the procedures that control the state machine flow (implemented in the **Main_Fsm.Fsm_Task** package). This design decision eliminates the generation of unneeded code and reduces the number of subprogram calls, making the application smaller and more efficient.

- Another private child package, named **Main_Fsm.States**, contains the definition of all the states of the state-machine as well as the specification of the procedures that should be executed when entering (`On_Entry` procedure), exiting (`On_Exit` procedure) or when staying (`Do_Activity` procedure) in the state. As in the case of the `Transition` concept above, the specification, empty body and corresponding subprogram invocations are generated depending on the value of the `hasOnEntry`, `hasOnExit` and `hasDo` properties of the `States`.

The body of the reactor task contains a **select** statement to perform an Asynchronous Transfer Control (ATC) [15] back to the reactor. In the case in which the reactor is executing the `Do_Activity` of the state that is going to be exited as a consequence of the received event, this ATC will abort the corresponding procedure, as the execution of the state machine artifact demands.

Finally, to end this section about the model-to-Ada transformation of a StateML$^+$ model, it only remains to explain the implementation of the different values of the `unexpectedTransitionPolicy` attribute of the `State-Machine` concept. As was said in section 2, UML 2.x says that the behavior of the state machine after receiving an event that does not trigger the fire of any of the transitions of the current state is undefined (UML calls this a "semantic variation point"). The StateML$^+$ completes this lack definition by offering the designer two possible alternative behaviors when state machine receives such a transition: it may silently ignore the unexpected transition or it may call a registered procedure to notify other parts of the program the occurrence of this condition.

In the later case, two new procedures, named `Subscribe_Handler` and `Erase_Handler`, are added to the generated **Main_Fsm.Fsm_Task** package in order to allow interested units to subscribe to the occurrence of unexpected transitions. These handlers will be sequentially called by the state machine when it detects such an unexpected transition.

4 Conclusions and Future Research

State-machines have been used in software system design since the early 1970s, being part of many software applications and, for their characteristics they are particularly useful in the embedded system domain. As a consequence, many tools have been developed to describe and generate executable code for these artifacts. Among these tools, probably one of the most widely used is STATEMATE [11].

The OMG adopted state-machines to describe the behavior of the elements that appear in their UML standard. As a consequence, new UML-compliant tools appeared in the marketplace allowing designers to use this artifact in their designs. However, as the scope of these tools is wider than just generating code for state-machines, they commonly produce complex and cumbersome code, making it difficult to extract the state machine code. In this sense, the main advantage of the MDE approach is that developers can decide the abstraction level, the scope of their applications and the way models are transformed into code, having full control over the development process.

This paper has presented an extended version of the previously developed StateML tool, aimed at designing non-hierarchical state-machines and generating the corresponding Ada code implementation. According to the research objectives outlined in the introduction of the paper, the improved StateML$^+$ meta-model now allows designers to model the behavior of the ACRoSeT abstract components and to generate the corresponding Ada implementation. Therefore, the work presented in this paper represents a first and decided step in the road to implementing a full MDE process for generating robotic applications.

The extended StateML$^+$ meta-model and tools now include many improvements over the previous versions, being these the most important: (1) the addition of regions, which enable the creation of hierarchical and concurrent state-machines, (2) a better implementation of the *run-to-completion* semantics associated to the state-machine artifact, and (3) the generated Ada code is thread-safe and ready for working in a multi-tasking environment. We are still working on some additional improvements in the following directions:

- To allow designers to define different concurrency policies for each state. This would allow them to decide whether the `do_activity` of a state should be executed in the Reactor or in a new thread created for this purpose.
- To implement and test alternative design patterns, such as the *Proactor* or the *Active Object* [8] ones, which may help improving the overall system flexibility, as explained in section 3.1.
- To define additional model-to-text transformations to different target languages. Finding alternatives to Ada can be a tough work, as there are not many languages providing such a good multi-task support.

References

1. Schmidt, D.: Model-Driven Engineering. IEEE Computer **39** (2006) 25-31
2. Bézivin, J.: On the Unification Power of Models. Software and Systems Modeling **4** (2005) 171-188.
3. Álvarez, B., Sánchez, P., Pastor, J.Á., Ortiz, F.J.: An architectural framework for modeling teleoperated service robots. Robotica **24** (2006) 411-418
4. Alonso, D., Vicente-Chicote, C., Sánchez, P., Álvarez, B., Losilla, F.: Automatic Ada Code Generation Using a Model-Driven Engineering Approach. In: 12th International Conference on Reliable Software Technologies, Ada Europe 2007, Vol. 4498. Springer, Geneva, Switzerland (2007) 168-179.

5. Szyperski, C.: Component Software - Beyond Object-Oriented Programming. Addison-Wesley / ACM Press (2002).
6. OMG: Object Constraint Language (OCL) Specification v2.0. The Object Management Group (2006).
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional (1995).
8. Schmidt, D., Stal, M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects. Wiley (2000).
9. The Eclipse MOFScript subproject. Available at: http://www.eclipse.org/gmt/mofscript/.
10. OMG: Unified Modeling Language: Superstructure v 2.0. The Object Management Group (2005).
11. Harel, D., Naamad, A.: The STATEMATE semantics of statecharts. ACM Transactions on Software Engineering Methodology **5** (1996) 293–333.
12. OMG: Meta-Object Facility Specification v2.0. The Object Management Group (2004)
13. Douglass, B.P.: Real Time UML: Advances in the UML for Real-Time Systems. Addison-Wesley Professional (2004).
14. Sendall, S., Kozaczynski, W.: Model transformation: the heart and soul of model-driven software development. IEEE Software **20** (2003) 42- 45
15. Burns, A., Wellings, A.: Concurrent and Real-time Programming in Ada 2005. Cambridge University Press (2007).