A Formal High-level Modeling Approach to Develop Reliable Components in Vision-based Robotics

Andrea Luzzana, Mattia Rossetti, and Patrizia Scandurra
Università degli Studi di Bergamo, DIIMM, 24044 Dalmine (BG), Italy
andrea.luzzana@unibg.it mattia.rossetti@unibg.it patrizia.scandurra@unibg.it

***Abstract*—This paper proposes the use of the *control-state Abstract State Machines* for a rigorous foundation in high-level modeling and validating component-based applications in *Vision-Based Robotics*. In particular, an extension of the classical flowchart notation for control-state ASMs is proposed to support modularization and reuse in a direct way. The resulting ASM models are to be intended as "ground models" that can be used as basis or patterns to practically model and formally validate the behavior of typical robotic control tasks, and to link (via successive refinements) these high-level models of components to their implementation code by making their functional correctness mathematically controllable. The proposed flowchart extension and the availability of reusable and validated ground models allow a better system design and speed up the development of the system.**

Keywords-components; abstract state machines; ground modeling; robotic control tasks.

I. INTRODUCTION

Vision-based robotics is a challenging research field [1]. One of the open and commonly stated problems in the field is the need for exchange of experiences, best practices, and high-level models of robust, reliable and flexible robot control applications with *visual servoing* functions.

Recently, we investigated [2] the use of the *Abstract State Machine* (ASM) formal method [3] for a systematic study and a rigorous foundation of modeling and validating *Visual Servoing* (VS) applications. The ASM method is a discipline for reliable system development, which allows to bridge the gap between informal requirements and executable code. The ASM formalism supports concurrency, heterogeneous state and modularity (compositional design and verification techniques). These features are essential to tailor *ground models* of control tasks definitions and associated synchronization/communication patterns of VS applications in rigorous, compositional and abstract terms. *Ground models* are blueprints of the to-be-implemented piece of "real world" that "ground the design in the reality" [4]. In particular, we exploit the notion of *control state ASMs* (a class of ASMs [3]) as a natural extension of Finite State Machines.

In this paper, we present an extension – called *pattern-oriented control-state ASMs* – of the classical flowchart notation for control-state ASMs to support *modularization* and *reuse* in a direct way. The proposed notation is useful to denote explicitly modeling elements to be further refined, to allow the definition/instantiation of recurring design solutions or patterns, to perform initial validation of separate high-level models and to improve model traceability between the flowchart diagrams and their concrete (textual) ASM specifications during the ground modeling and development process. In this context, the term "pattern" is to be intended to have its classical meaning, i.e., as a schema of a recurring solution, rather than the meaning of "design pattern" as in the book of the GoF (Gang of Four).

We here repeat our previous experience in the embedded system-on-a-chip domain [5][6] to shift the focus from implementation to design through high-level modeling. Our approach combines the expressive power and accuracy of control state ASMs with the intuition provided by visual flowchart descriptions to capture the behavioral view of task-level control of VS applications. As starting point, we manually extracted from the structure of existing (basically C/C++) code architectural descriptions (in terms of UML component diagrams) of high-level models of component control tasks and also recurring synchronization/communication patterns between tasks that could be used for the ground modeling and analysis in ASM. We then defined these ASM abstract models using the pattern-oriented control-state ASM notation, transformed such models into executable ASM models using the notation ASMETA/AsmetaL [7][8], and then validated them through basic formal analysis techniques (simulation and scenario-based simulation) [9]. The resulting and validated ASM models are used as basis or patterns for high-level modeling and validating typical control tasks of VS applications in a formal way, thus leading from the abstract models to executable (C/C++) code by making their functional correctness mathematically controllable.

This paper focuses on presenting the extended control-state ASM notation and it is organized as follows. Section II introduces the reader to the field of vision guided robotics by illustrating the synchronization/communication issues at control task-level covered in the paper. Section III provides background notions on the ASMs. Section IV presents the pattern-oriented control-state ASMs. Section V presents the ASM ground modeling of some control tasks synchronization/communication patterns and, as major case study, their instantiation to model a VS application. Section VI provides some details on implementation issues. Section VII presents

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-230-1

some related works. Finally, Section VIII concludes the paper and sketches future directions.

II. Application Domain

This section discusses about modeling issues in visual assisted robotic control architectures, with emphasis on synchronization/communication of control tasks.

Robot control applications with VS features: The main task of a robot automatic control system is to drive robot actuators (typically electrical motors) in order to follow a trajectory passed to the controller by higher level applications of the control architecture, such as motion planners, production cell controllers, visual systems, etc.

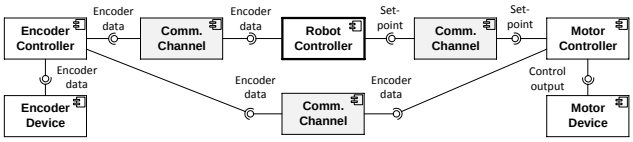


Figure 1. A Robot controller architecture.

The UML component diagram in Figure 1 shows a typical architecture of a robot control application. A robot controller sends set-point values to a motor controller. The encoder controller acquires data from encoders in order to send them to both the robot controller and the motor controller. This last task has to read also the data (motors set-point) produced by the robot controller in order to drive motors. The motor and the encoder controllers communicate directly with their respective devices through physical interconnections. This control scheme presents non-trivial issues related to the timing and the synchronization of the involved controllers since all the tasks have to exchange data periodically.

The need for strict periodic tasks makes it necessary to encapsulate the communication functionality in a separate component, the *communication channel*, thus decoupling the producer (or sender) from the consumer (or receiver). A communication channel can be implemented on top of several protocols. For data exchange among tasks, real-time programming guides (e.g., [10]) typically suggest the use of a *shared memory* (e.g., a FIFO, a ring buffer, a stack, etc.) in order to ensure lower memory usage and better performance. In this work, we will focus on the use of the *Swinging Buffer* [10] (described below) as shared memory.

A robot builds a representation of the surrounding environment by acquiring data from several sensors. *Visual servoing* [1] is a technique that uses feedback information extracted from a vision sensor to control the motion of a robot. A closed-loop control of a vision-based robotic system usually consists of two intertwined processes: tracking and control. This architecture (see Figure 2) performs the control of the robot in two separate stages: first, the vision system provides input to the robot controller by acquiring and elaborating images; then, the robot controller uses joint feedback to internally stabilize the robot. Optionally, set-point computation can require the acquisition of robot data via another communication channel.

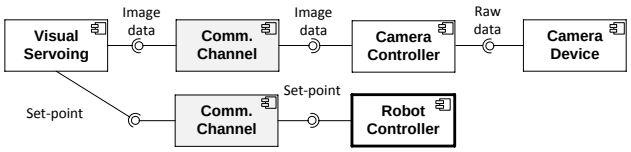


Figure 2. A Visual Servoing Robot architecture.

Using parallelism can improve the performance of the system, but it introduces new non-trivial issues due to VS functions. First, the visual servoing task can be exploited only at the presence of an image, so it is the only non-periodic (asynchronous) task in the control scheme. Second, the time required for elaboration is not constant because computer vision algorithms efficiency is strongly affected by aspects like quality of images, complexity of the environment and so on. Finally, the visual servoing task is both a producer and a consumer because it requires images from camera for its elaboration and then needs to transmit the set-point information to the robot.

Synchronization and Communication issues: Basically, control tasks can be classified in *asynchronous* and *synchronous*. Asynchronous tasks are data-driven, because their elaboration starts when there are data to be consumed and ends with data transfer. Synchronous tasks are, instead, time-driven, as they are periodic and have deadlines to respect. Figure 3 summarizes the possible communication types that we cover in our work, as collected from visual servoing and robot control applications (such as pick and place, object tracking and micro-assembly). The analyzed solutions involve the use of swinging buffers for the communication between tasks operating at different frequencies. A swinging buffer can be viewed as an advanced circular buffer using two or more shared memory arrays instead of the single array adopted by a circular buffer. While the producer task fills up one of the buffers, the consumer empties another one. When a task reaches the end of the buffer that it is using, it starts operating from the beginning of another unused array. Since tasks works on different memory locations, no lock for the mutual exclusion is needed to access to the data on the buffer, but only for updating the read/write indexes.

Producer \ Consumer	Asynchronous	Synchronous
Asynchronous	**Asynchronous Message Passing** typical Producer-Consumer	**Swinging Buffer Communication** Visual Servoing TO Robot Controller
Synchronous	**Swinging Buffer Communication** Robot Controller TO Visual Servoing	**Swinging Buffer Communication** Sensor TO Robot Controller TO Motor

Figure 3. Task communication types.

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-230-1

III. Background on ASMs

ASMs are an extension of Finite State Machines (FSMs) [3] where unstructured control states are replaced by states of arbitrary complex data. The *states* of an ASM are multi-sorted first-order structures, i.e., domains of objects with functions and predicates (boolean functions) defined on them. The *transition relation* is specified by named rules describing how functions change from one state to the next. A transition rule has the basic form of *guarded update* "**if** *Condition* **then** *Updates*" where *Updates* is a set of function updates of the form $f(t_1, \ldots, t_n) := t$, which are simultaneously executed when *Condition* is true. f is an arbitrary n-ary function and $t_1, \ldots, t_n, t$ are first-order terms. Essentially, to fire this rule in a state S_i, $i \geq 0$, evaluate all terms $t_1, \ldots, t_n, t$ at S_i and update f to t on parameters $t_1, \ldots, t_n$. This produces another state S_{i+1}, which differs from S_i only in the new interpretation of f. A set of *rule constructors* allows to express simultaneous parallel actions (`par`), sequential actions (`seq`), iterations (`iterate`, `while`, `recwhile`), and submachine invocations returning values. Non-determinism (existential quantification `choose`) and unrestricted synchronous parallelism (universal quantification `forall`) are also supported.

Control-state ASMs are a class of ASMs used to model some overall status or mode, guiding the execution of guarded synchronous parallel updates of the underlying state. Figure 4 (slightly adapted from [3]) shows on the left the conventional flowchart notation that include three basic symbols. It also shows on the right the corresponding ASM rule scheme (in textual notation) for control state ASMs. Circles denote phases (also called control states or internal states), hexagons (optional) denote test predicates (also called conditions or guards), and rectangles denote update actions (i.e., application of ASM rules, including rule invocations of submachines) and are also optional. The finitely many control states *ctl_state* $\in \{1 \ldots m\}$ are used to describe different system modes.

IV. Pattern-oriented Control State ASMs

We extended the flowchart notation of control state ASMs to better enhance some aspects and to capture new ones. Precisely, we defined a pattern-oriented extension of the classical flowchart notation for control state ASMs to denote explicitly parts of the models that have to be further refined, to denote the definition/instantiation of a pattern, and to improve model traceability by allocation links between the flowchart diagrams and their concrete implementations (i.e., the ASM specifications) in the textual language AsmetaL.

Revised symbols. We adopt (see Figure 5) dashed lines for guards and actions to indicate that these elements require further refinement, i.e., the test predicate for the guard and the rule for the action. For example, an action can be refined by introducing other action-state-condition blocks to model the intended activity. The optional text {`text`}

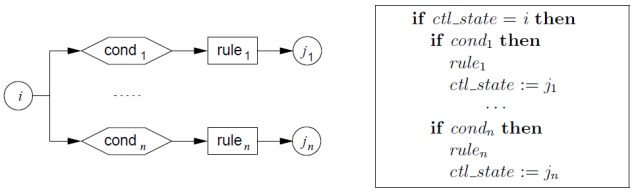


Figure 4. Control state ASMs.

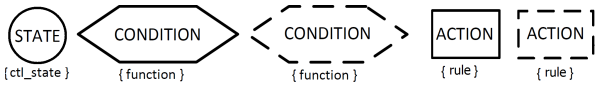


Figure 5. Extended control state ASM notation.

near a graphical symbol is used to link diagrams to their concrete ASM specification `spec` (in our case, AsmetaL specification). Specifically: for a state symbol it denotes the function name in the `spec` representing the underlying control state variable; for a guard symbol it denotes the test predicate name in the `spec`; for an action symbol it denotes the rule name in the `spec` implementing it.

Pattern machines. For modularization and reuse purposes, we introduce two new symbols (see Figure 6) denoting the concepts of *pattern* (pattern definition) and of *pattern instantiation* blocks, respectively. A pattern block is to be intended as a placeholder for a recurrent and complex action block, also referred to us as *pattern machine*. Figure 6 shows the shape of such a pattern machine that includes an entering arrow followed by (at least) an action-state-condition block closed with a floating exit arrow. A pattern machine consisting of an action block only is also admitted. The circles represent the internal states of the pattern machine and it usually requires a fresh control state variable `ctl_state`. It is a piece of reusable ASM model that can be validated and verified separately and then re-used in more complex ASM specifications. The entering arrow denotes always the evaluation of the guard `isUndef(_ctl_state)` that is the mandatory condition that enables the execution of the pattern machine. The floating exit arrow denotes the exit point and implies always the mandatory update `ctl_state := undef`.

A pattern machine is then specified in terms of a named rule and this rule will occur as subrule of the containing machine. Moreover, we assume that (otherwise specified) the pattern machine is composed with the other occurring rules (action blocks), if any, of the containing machine according

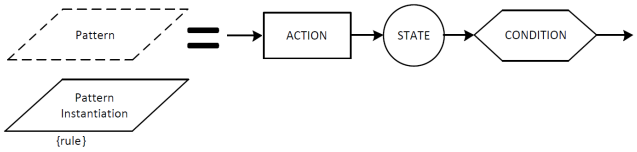


Figure 6. Pattern notation.

to the synchronous parallelism semantics of the **par**-rule. The rule name of the pattern machine can be specified (see Figure 6) near the pattern instantiation symbol.

V. Ground Modeling Robot Control Tasks

We use pattern-oriented control state ASMs for specifying ground models of robot control tasks.

A. Producer/Consumer models

Figure 7 shows the control state ASM of a simple data producer module. It contains a pattern machine for the writing operation on the swinging buffer, presented in Section V-B. The first action of a producer task is to wait for a trigger in order to start to acquire and elaborate data; lastly, data elaborated have to be written in a shared memory (in our case, a swinging buffer). This machine has blocks to be refined, because their specific behavior depends on the tasks nature: tasks can be asynchronous or synchronous, and in turn master or slave. Listing 1 reports, using the AsmetaL notation, a possible definition of the test predicates for the events *Trigger* and *Elaboration Time Elapsed* (their names are also denoted in the diagrams). The event *Trigger* may require further refinement. The listing reports a possible definition of this test predicate (in a particular phase of the specification development process) that captures two cases: it is the boolean OR of "waiting for a period of time" for a synchronous periodic task, and of an "always-true" condition for an asynchronous task. Both the test predicates take as parameter the scheduler `c`, since in our applications each component is associated to and managed by a scheduler. The current task scheduled on `c` is represented by the function `scheduler_MainThread(c)`.

The data consumer model is straightforward.

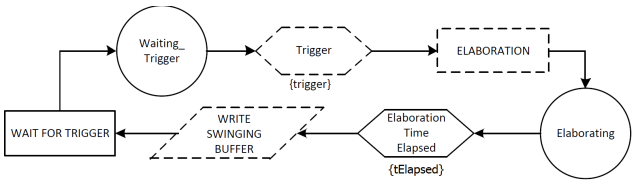


Figure 7. Data Producer.

Listing 1. Producer/consumer test predicates.

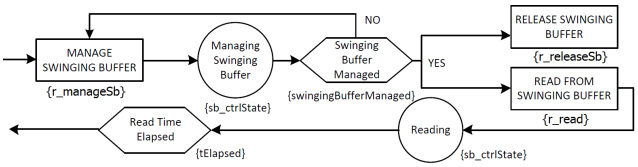
```
function trigger($c in Scheduler) = (task_kind(
     scheduler_mainThread($c)) = SYNCHRONOUS and
     task_elapsedTimeOfPeriod(scheduler_mainThread($c)) >
     time(scheduler_currentPhase($c)))
      or task_kind(scheduler_mainThread($c)) =
           ASYNCHRONOUS

function tElapsed($c in Scheduler) =
      scheduler_currentScheduleTime($c) > time(
           scheduler_currentPhase($c))
```

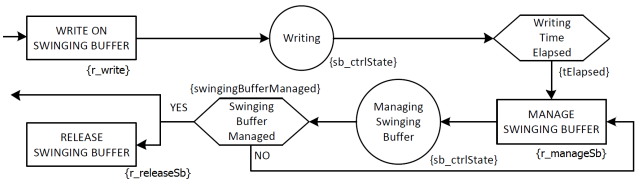
B. Swinging Buffer reading/writing models

The swinging buffer introduces the concept of multiple shared memory areas. Hence, the producer and consumer tasks do not share a memory, but only the read/write pointers to different memory areas. To avoid overwriting problems, only one task, the *master task* (either the producer or the consumer), can manage the indexes update, while the other tasks, the *slaves*, behave just classical producers/consumers. However, in real-time tasks, also the nature of the communicating tasks and their frequency must be considered. In the case of two asynchronous communicating tasks, a producer and a consumer, the master task can be either the producer or the consumer. In the case of two synchronous tasks, instead, a common practice is to set the slower task as the master one. In fact, as there is not a single shared memory area, the master task has to get the lock only when it has to update its data pointers. Moreover, during its elaboration, it does not need to get the lock because, as the slave tasks do not manage indexes, there is no possibility of an inconsistent update. Finally, in the case of a synchronous-to-asynchronous communication, the asynchronous task is the master slave because it does not carry out data elaboration periodically, but only when new data are available.

This behavioral variability is captured by the ASM pattern machines for reading/writing from/to swinging buffer shown in the Figures 8, 9 and 10. In particular, as an example, Listing 2 for an *Asynchronous-Master-Writing* operation (see Figure 8-b) reports the corresponding ASM specification using the AsmetaL notation. The AsmetaL implementation of the other pattern machines can be found in [11].



(a) Reading.



(b) Writing.

Figure 8. Swinging Buffer - Asynchr. Master Read/Write.

The *Asynchronous-Master-Writing operation* (see Figure 8-b) implies first to write data on the shared memory directly (state *Writing*), without acquiring the lock for the critical section. When the writing operation is terminated (after a certain period of time has passed), the swinging buffer pointers have to be updated in order to signal to the consumer that new data are ready. In the next state *Managing*

Swinging Buffer, the asynchronous task tries (by the iterative flowchart part) to get the lock to the critical section for updating the pointers by executing the action *MANAGE SWINGING BUFFER*. After the pointers are updated, the control exits by releasing the lock (*RELEASE SWINGING BUFFER*). As an example, Listing 3 reports the definition of the rule *MANAGE SWINGING BUFFER* using the AsmetaL notation; note again that this rule captures the behavior of both master and slave tasks by distinguishing clearly these roles, since it is reused also as action in the other pattern machines. The AsmetaL implementation for the other actions and test predicates can be found in [11].

The *Asynchronous-Master-Reading pattern* (see Figure 8-a) is similar to the writing one, despite of action order. In fact, in this case the *Managing Swinging Buffer* action has to be performed before reading.

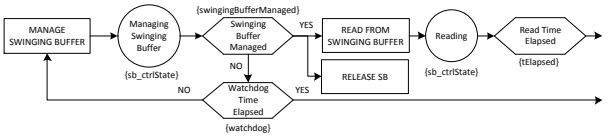
Listing 2. Asynchronous-Master-Writing pattern machine.

```
rule r_async_master_writing ($c in Scheduler) =
par
     //Entering into the pattern machine
 if (isUndef (sb_ctrlState(scheduler_currentPhase($c))))
 then par
    r_write [$c, scheduler_currentPhase($c)]
    sb_ctrlState(scheduler_currentPhase($c)):= WRITING
 endpar
 if (sb_ctrlState(scheduler_currentPhase($c)) = WRITING)
 then if (tElapsed ( $c))
         then par
         r_manageSb [$c, scheduler_currentPhase($c)]
         sb_ctrlState(scheduler_currentPhase($c)):=
              MANAGING_SB
              endpar
 if (sb_ctrlState(scheduler_currentPhase($c)) =
      MANAGING_SB) then
      if (swingingBufferManaged($c)) then //Index updated
            par
             r_releaseSb[$c]
             // Exit from pattern machine
             sb_ctrlState(scheduler_currentPhase($c)):=
                  undef
            endpar
      else //Index not updated
            par //Try again to acquire the lock
             r_manageSb [$c, scheduler_currentPhase($c)]
             sb_ctrlState(scheduler_currentPhase($c)):=
                  MANAGING_SB
            endpar
endpar
```

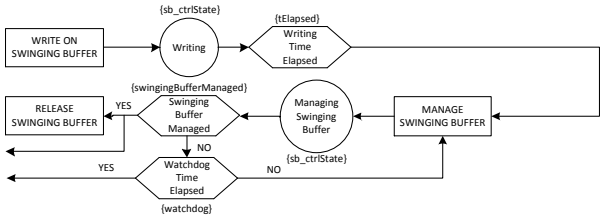
For the reading/writing operations of synchronous master tasks (see Figure 9), the transition for managing the swinging buffer indexes is slightly modified: if the lock has not been acquired, a further guard *watchdog Time elapsed* is evaluated: if false (there is no time), the task skips the index update phase and exits. Finally, for synchronous slave tasks (see Figure 10 for the reading operation) if the lock cannot be acquired, the task simply skips the reading/writing action. The writing operation is fully similar.

Listing 3. The MANAGE SWINGING BUFFER rule.

```
rule r_manageSb($c in Scheduler, $phase in SchedulePhase)=
par
      if (syncPriority($phase) = MASTER)
      then seq
            r_SwingingBuffer_getLock [sharedMemory($phase)
                 , scheduler_mainThread($c)]
            r_SwingingBuffer_updateIndexes [sharedMemory(
                 $phase) , scheduler_mainThread($c)]
      endseq
      if (syncPriority($phase) = SLAVE) then
            r_SwingingBuffer_getLock [sharedMemory($phase)
                 , scheduler_mainThread($c)]
endpar
```



(a) Reading.



(b) Writing.

Figure 9. Swinging Buffer - Synchr. Master Read/Write.

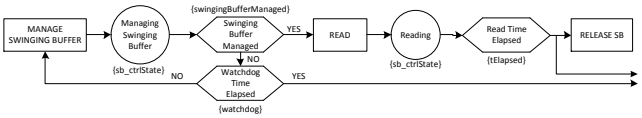


Figure 10. Swinging Buffer - Synchronous Slave Read.

C. Major case study: A VS application model

Let us consider the VS application case study described in Section II. Figure 11 shows the ASM control state for the visual servoing component shown in Figure 2. It elaborates images in order to produce the commands to send to the robot controller component. So, it is an asynchronous task because it can elaborate images only when they are available. Note that it plays the role of both an asynchronous producer and an asynchronous consumer: it is an asynchronous consumer of images coming from the synchronous camera controller component and a producer for the synchronous robot controller. It communicates with two synchronous tasks through two swinging buffers and, being asynchronous, it is the master.

The AsmetaL implementation of this major case study can be found at [11].

VI. Tool-support and Target Implementation

We adopt the ASM modeling and analysis toolset ASMETA [7], based on the Eclipse EMF. A graphical editor for the flowchart extension presented in this paper is being developed. For the execution of ASM models written in

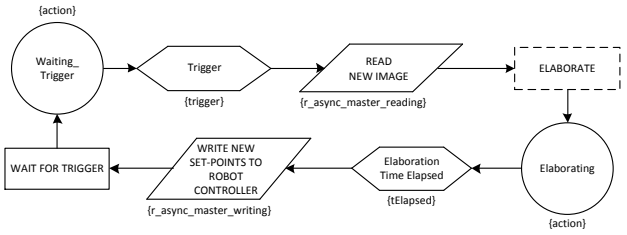


Figure 11. Visual Servoing component.

AsmetaL we use the ASM simulator *AsmetaS*. We also exploit the tool *AsmetaV* to run execution scenarios and report any violation of the expected behavior.

As target implementation platform, we have been working on generating automatically C/C++ code from the ASM models through a mapping towards common used VS libraries, including: *Posix Threads* library, *dc1394* library for managing the communication from/to a firewire (IEEE1394) [13] camera, and *OpenCv* for image elaboration.

VII. Related Work

The use of formal methods to describe the underlying data exchange mechanisms in distributed control systems is not a common practice. Within the ASM community, some few works exist related to the ASM modeling of multi-process synchronization problems and of inter-process communication problems [14][15][16]. We took inspiration from all of them. In particular, to compile ASMs into C/C++ code, we have been repeating the previous experience in the FALKO project (a tool for railway simulation) [12].

The work in [17] reports on the development of AsmL (an ASM notation from Microsoft Research) specifications of Synchronous Dataflow domain schedulers of Ptolemy II – simulation and code generation framework for heterogeneous, concurrent, real-time embedded systems [18]. Their goal is to give a precise semantics to the implementation. The use of the ASMs to design and verify low level communication and data exchange mechanisms, however, seems lacking.

VIII. Conclusion and Future Work

We proposed an ASM-based easy and scalable approach to design reliable and reusable ground models of control tasks and communication/synchronization mechanisms in VS robotic applications. The proposed flowchart extension for control-state ASMS and the availability of such reusable and validated ground models allow a major comprehension of the system design (even to non ASM experts) and speed up the development itself. Though our work is targeted to the VS domain, we believe the approach can be easily extended to a wide range of real-time applications.

As future work, we aim at defining an ASM model library for VS applications and using complex analysis tools in the ASMETA toolset for formal verification.

Acknowledgment

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. FP7-ICT-231940-BRICS (Best Practice in Robotics).

References

[1] S. Hutchinson, G. D. Hager, and P. I. Corke. A tutorial on visual servo control. *IEEE Transactions on Robotics and Automation*, Vol. 12, pp. 651–670, 1996.

[2] A. Luzzana, M. Rossetti, P. Righettini, and P. Scandurra, Modeling Synchronization/Communication Patterns in Vision-Based Robot Control Applications Using ASMs. In *ABZ Conf.*, 2012, pp. 331-335.

[3] E. Börger and R. Stärk, *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.

[4] E. Börger, Construction and analysis of ground models and their refinements as a foundation for validating computer-based systems. *Formal Asp. Comput.*, Vol. 19, pp. 225–241, 2007.

[5] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio, A SoC Design Methodology Involving a UML 2.0 Profile for SystemC. In *Proc. of Design, Automation and Test in Europe Conf.*, pp. 704-709, 2005.

[6] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio, A model-driven design environment for embedded systems. In *Proc. of the Design Automation Conf.*, pp. 915-918, IEEE Press, 2006.

[7] The ASMETA website. `http://asmeta.sf.net/`, [retrieved: October, 2012].

[8] P. Arcaini, A. Gargantini, E. Riccobene, and P. Scandurra, A model-driven process for engineering a toolset for a formal method. *J. Softw., Pract. Exper., Vol. 41, pp. 155-166, 2011.*

[9] A. Gargantini, E. Riccobene, and P. Scandurra, A Metamodel-based Language and a Simulation Engine for Abstract State Machines. *J. UCS, Vol. 14, pp. 1949–1983, 2008.*

[10] H. Bruyninckx, Real-time and embedded guide. *KU Leuven, Mechanical Engineering*, 2002.

[11] ASM RTpattern library. `http://asmeta.svn.sf.net/viewvc/asmeta/asm_examples/RTPatternLibrary/`, [retrieved: October, 2012].

[12] J. Schmid, Compiling abstract state machines to C++. *J. UCS*, Vol. 7, pp. 1068–1087, 2001.

[13] "IEEE Standard for a High-Performance Serial Bus". IEEE Std. pp. 1–906, 2008, doi:10.1109/IEEESTD.2008.4659233.

[14] E. Börger and I. Craig, Modeling an operating system kernel. In *Informatik als Dialog zwischen Theorie und Anwendung*, pp. 199–216. Vieweg+Teubner, Wiesbaden, 2009.

[15] R. F. Stärk and E. Börger, An ASM specification of C# threads and the .NET memory model. In *Abstract State Machines*, LNCS 3052, pp. 38–60. Springer, 2004.

[16] W. Müller, J. Ruf, and W. Rosenstiel, An ASM based SystemC simulation semantics. *SystemC: methodologies and applications*, pp. 97–126, 2003.

[17] D. L. Cuadrado, P. Koch, and A. P. Ravn, ASML specification of a Ptolemy II scheduler. In *Abstract State Machines*, LNCS 2589, page 417. Springer, 2003.

[18] PTOLEMY website. `http://ptolemy.eecs.berkeley.edu/`, [retrieved: October, 2012].

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-230-1