Project: COMPASS

Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

COMPASS

SysML State Machines: a formal model for refinement

COMPASS White Paper WP03

July 2013

Public Document

http://www.compass-research.eu

Authors:
Alvaro Miyazawa, Ana Cavalcanti, University of York, UK

Abstract:

This white paper follows on from COMPASS WP02 (SysML Blocks in CML) to present a denotational semantics of SysML state machine diagrams using CML. We discuss how this semantic formalisation can be used in the wider context of a comprehensive semantics for SysML models. In particular, we focus on the relationship between block definition and state machine diagrams, based on a modelling pattern where a state machine diagram is used to define the possible behaviours of a block. We define the semantics of an extensive subset of state machine constructs, the choice of which is based on actual usage in examples and case studies.

The work is currently being used in the COMPASS project as a basis for two main lines of investigation: integration and refinement. On the integration front, we currently have a formal semantics of sequence and activity diagrams as well as of SysML blocks. In the second line of work, we are interested in lifting the CML notion of refinement to support refinement at the level of SysML, as well as the development of refinement strategies for verification of implementations.

SysML State Machines: a formal model for refinement

Alvaro Miyazawa Ana Cavalcanti

July 2013

1 Introduction

SysML [OMG10] is an extension of UML 2.0 to support modelling for systems engineering. SysML retains a number of UML 2.0 diagrams (activity and sequence diagrams), modifies others (block definition, internal block, and state machine diagrams) and adds a new type of diagram (parametric diagrams). It supports modelling of a variety of aspects of a system, including software and hardware components and socio-technical aspects. Whilst it is difficult to gauge adoption of SysML in industry, its current support by tool vendors such as IBM [IBM], Atego [Ate] and Sparx Systems [Spa] indicates that adoption is perceived as wide.

Our aim is to support the application of formal analysis tools and techniques at the level of the graphical notations used in current industrial practice. In particular, we discuss our results on bridging the gap between SysML and a formally grounded specification language that supports a variety of analysis techniques.

Whilst SysML is a graphical notation, the COMPASS modelling language, CML, builds on well-known formal specification languages, namely VDM [FL09] and CSP [Hoa85]. Its approach to modelling reactive behaviour and its semantic model are those adopted in the *Circus* [CSW03] family of refinement languages. The semantics of CML and *Circus* use the Unifying Theories of Programming to cater for object-orientation [SCS06], time [SCJS10], as well as synchronicity [BG09], for instance. *Circus* has been successfully used in practical applications [CCO11, MC12].

We present a denotational semantics of SysML state machine diagrams using CML, and discuss how this formalisation can be used in the wider context of a comprehensive semantics for SysML models. In particular, we focus on the relationship between block definition and state machine diagrams, based on a modelling pattern where a state machine diagram is used to define the possible behaviours of a block. We define the semantics of an extensive subset of state machine constructs, the choice of which is based on actual usage in examples and case studies.

Formal semantics for many variants of state diagram notations are available in the literature. In most cases, key features (e.g., interlevel transitions) are not covered and strong restrictions are imposed to produce simpler models, making extensions of the approach difficult. In addition, these semantics are usually addressed in isolation, without consideration to the usage of state machine diagrams in larger models. Finally, most of them are not suitable for refinement. We are interested in the use of refinement as a means of formally capturing the life cycle of a system, from development through deployment and execution. Refinement supports analysis with respect to previous models as well as stepwise transformation and controlled evolution. Our main contribution is a semantics of SysML state machine diagrams that is appropriate for refinement and further integration with models of other SysML diagrams.

2 Overview of our CML models

The model of a state machine diagram belonging to a block **B** is a parametrised CML process that accepts communications through a number of external channels. The channel `B_addevent` allows new events to be added to the event pool. Channels whose name are suffixed by `_op` allow the state machine to return operation results, through `B_op`, and request operations from other blocks through other `_op` channels. Channels whose names have the suffix `_sig` allow the state machine to send signals to other blocks, and channels whose names start with `B_set_` and `B_get_` allow access to the state components of the process that models **B**. The parameter of the process is used to identify the instance of the block to which the state machine belongs.

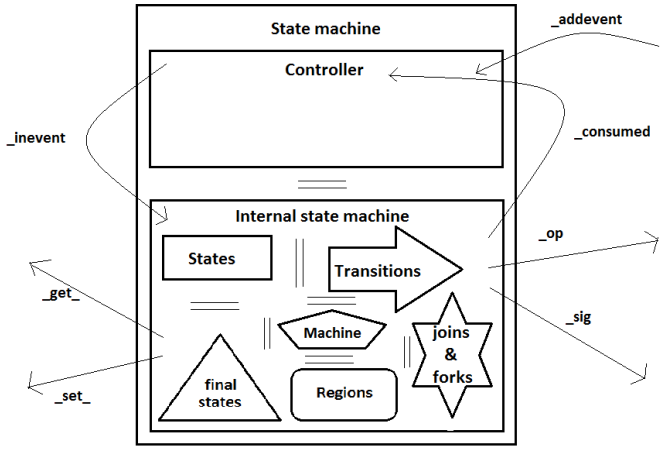

Figure 1: Overview of the model of state machines.

Figure 1 gives the overall structure of the CML processes that model state machines. Each of them is defined by a parallel composition of two other process: the first, called the controller process, handles events and have state components that model the event pool and the deferred event queue, the second, called the internal process, models the execution of the state machine under a particular event. They communicate with each other through the internal channels `B_inevent` and `B_consumed`. `B_inevent` allows the controller to send events to the internal process (from the event pool or deferred event queue), and `B_consumed` allows the controller to ask the internal process if the event was consumed, not consumed or deferred. The isolation of event management within the controller process allows us to model aspects of event management that are usually overlooked in other formalisations, namely, deferred events and their interactions with the event pool and the state machine itself.

The communication protocol between the two processes is as follows. The controller selects an event from either the pool or the queue and sends it to the internal process through the channel `B_inevent`. The internal process then processes the event and communicates through `B_consumed` one of the values: `true`, `false` or `<defer>`. The first value indicates that the event triggered a change of state in the state machine, the second that the event was accepted but did not trigger a state change, and the third that the event was deferred. In the case of deferral, the controller adds the event to the queue of deferred events.

The internal process receives an event through the channel `B_inevent`, processes it as specified by the state machine it models, and provides the result through `B_consumed`. The processing of an event may lead to communications over the channels `B_set_` (to read the state of the block's process), `B_get_` (to modify the state of the block's process), `_op` (to answer and send operation calls) and `_sig` (to send signals).

Whenever the event pool is not empty, the controller starts a cycle by sending an event from the pool to the internal process. If it is consumed, the controller sends each of the deferred events to the internal process. Since an event was consumed, a change of state may make it possible to consume a deferred event.

The controller and the internal processes are both parametrised by the identifier of the instance of the block. The internal process is stateless, and its main action is the parallel composition of actions that model the states, regions, final states, non-completion and completion transitions, join and fork pseudostates, and a machine action that models the top-level behaviour of the machine.

Whilst most aspects of state machines have been covered in other formalisations, to the best of our knowledge, most of these formalisations are operational, aiming at execution and possibly model checking, but frequently not suitable for other forms of verification. In our work, we take a different approach by proposing a denotational semantics where the granularity of the state machine is preserved as much as possible, with the main elements modelled by CML actions and their composition modelled by parallelism.

The models of the elements of the state machine include protocols that specify how they interact with each other. This level of granularity allows us to focus on particular elements or subsets of elements when analysing the model. The protocol for the actions that form the main action of the internal process follows

the structure of the state machine.

The machine action controls the execution of all other actions. It has two distinct phases of behaviour: initialisation and execution. In the initialisation, it requests the top regions of the state machine to be entered, which then request their substates to be entered and so forth, until a stable configuration (that is, one in which all regions of an active state are active, and exactly one substate of each active regions is active) is reached. The execution consists of a loop in which at each step the machine action reads an event from the controller, sends it to the actions that model the top regions, evaluates the result of processing of the event, sends the result to the controller, triggers the enabled transitions (that is, transitions whose trigger contains the event being processed and whose guard evaluates to true) and waits for the transitions to finish executing. The pattern of interactions between the actions that model states and regions is similar.

The actions that model transitions whose guard and event are true become enabled and prepare to execute or be cancelled by the action that models the source state of the transition (if a conflict occurs). The actions of transitions that are not enabled wait to restart. The action that models an active state requests that the actions for the enabled transitions cancel their execution until only one remains, in which case, it reports the consumption of the event. If none of the transition actions are prepared to execute, the state action reports that the event has not been consumed.

The actions that model states, regions and final states communicate through the channels `B_inevent`, `B_consumed`, `enter`, `entered`, `exit`, `exited` and `instate`. The first allows the action to receive an event, the second to respond to the event, the third to enter the element that it models, the fourth to acknowledge that it has been entered, the fifth to exit it, the sixth to acknowledge that it has been exited, and the seventh to communicate the status of the element. Additionally, the models of final states communicate on `final_state` to indicate a final state has been reached. The models of states can communicate on the channel `enabled` to ask whether an event has been processed by actions modelling transitions, `final_state` to check whether its internal behaviour has completed, and `completion` to indicate that completion transitions can execute.

The models of transitions, join and fork pseudostates communicate on `B_inevent`, `B_enabled`, `enter`, `entered`, `exit`, `exited` and `cancel`. The first two are used to send events and the result of their processing, the third and fourth are used to request the activation of a state and wait for its successful activation, the fifth and sixth channels are used similarly to request and wait for the deactivation of a state, and the final channel is used to request the cancellation of conflicting transitions. The models of completion transitions and join pseudostates additionally wait on `completion` before executing.

Whilst all formalisations cover some form of state and transition, they often impose restrictions over the structure of these elements, such as not allowing interlevel transitions. Moreover, join and fork pseudostates are frequently omitted. In our formalisation, we consider the transitions connected to join and fork pseudostates part of these pseudostates. For this reason, the models of join and fork pseudostates are similar to those of transitions, since they lead to states being exited and entered, but are specialised to allow multiple (orthogonal) states being entered or exited.

3 Conclusions

In this paper, we have proposed a formalisation of SysML state machine diagrams based on the CML notation. We adopt a compositional approach based on the use of parallelism as conjunction of requirements. It generates formal models whose components can be easily traced back to components of the diagram. Moreover, the translation rules can also be taken as mathematical functions and used to encode the state machine notation in a theorem prover.

Our treatment of state machine diagrams covers a large number of elements, namely, simple, orthogonal and non-orthogonal states, regions, junctions, completion and non-completion transitions, join and fork pseudostates and final states. Besides enabling the use refinement to reason about state machines, we have also uncovered some open issues related to the semantics of state machine diagrams. In all cases, we have taken pragmatic decisions based on available examples to enable the production of meaningful models.

First of all, we have identified guidelines of usage that take into account the joint use of the main SysML structural and behavioural blocks. We have presented here part of those guidelines affecting state machine diagrams, and a complete account can be found in [ACC+13].

Since the action language of SysML state machines is not specified, we have adopted the CML data

notation. This facilitates the use of CML tools to reason about SysML, but has also uncovered the issue of shared access to block data.

Additionally, we support the use of state machine diagrams to specify the behaviour of a block operation as commonly adopted in examples [FMS11]. We do not, however, support the use of recursion in operations defined by state machine diagrams because SysML state machines do not support recursive execution. In order to support recursion, a mechanism such as local event broadcasts [Mat] in MATLAB's Stateflow would be required; however, this mechanism significantly increases the complexity of the semantics due to the possibility of inconsistencies raised by the recursive execution of the state machine. A detailed formalisation of the local event broadcast features of Stateflow can be found in [MC12].

We have validated the translation strategy through the manual translation of examples, simulation and analysis using CSP tools such as ProBE [For98] and FDR2 [For99]. A tool that automatically generates CML models of state machine diagrams is currently under development.

Our semantics can be extended to cover other features. For instance, shallow and deep history pseudostates can be supported by modifying the rules that specify the `off` action of a state (that contains a history pseudostate, and its descendants in the case of deep history) and the status of the state. A record of the last active substate must be kept and used when entering a state to activate the appropriate substate.

A choice pseudostate is similar to a junction, but the transitions from a choice pseudostate use the state of the block after the execution of its transitions. Whilst choice pseudostates are not directly supported, their behaviour is identical to a simple state without action or internal transitions. Since the transitions leaving a choice pseudostate cannot have triggers, they become effectively completion transitions. Furthermore, since the transitions must form a cover (i.e., the conjunction of the guards must be true), it is always the case that at least one transition is enabled and is executed. We can treat choice pseudostates directly in the semantics by ascribing the semantics of simple (action-less) states to the choice pseudostates, or we can perform a model to model transformation where choice pseudostates are replaced by simple states.

The current work is being used as a basis for two main lines of investigation: integration [ACC+13] and refinement. On the integration front, we currently have a formal semantics of sequence and activity diagrams as well as of SysML blocks. The translation of block definition diagrams and internal block diagrams provides us with a common ground for the integration of the other models.

We are also working on a refinement strategy that collapses part of the parallelism in the models, reducing the amount of communication and potentially the number of states for model checking. This is part of our second line of work, in which we are interested in lifting the CML notion of refinement to support refinement at the level of SysML, as well as the development of refinement strategies for verification of implementations.

References

[ACC+13] Lucas Albertins, Ana Cavalcanti, Marcio Cornélio, Juliano Iyoda, Alvaro Miyazawa, and Richard Payne. Final Report on Combining SysML and CML. Technical Report D22.4, COMPASS Deliverable, March 2013.

[Ate] Artisan Studio. `http://www.atego.com/products/artisan-studio/`. Accessed: 11-04-2013.

[BG09] Andrew Butterfield and Paweł Gancarski. The Denotational Semantics of slotted-*Circus*. In Ana Cavalcanti and Mads Damm, editors, *Formal Methods 2009*, volume t.b.a. of *LNCS*, Eindhoven, Netherlands, November 2009. Springer.

[CCO11] Ana Cavalcanti, Phil Clayton, and Colin O'Halloran. From control law diagrams to Ada via *Circus*. *Formal Aspects of Computing*, pages 1–48, 2011. 10.1007/s00165-010-0170-3.

[CSW03] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2 - 3):146 — 181, 2003.

[FL09] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edition, 2009. ISBN 0-521-62348-0.

[FMS11] S. Friedenthal, A. Moore, and R. Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011. 2nd edition.

[For98] Formal Systems (Europe) Ltd. *Process Behaviour Explorer - ProBE User Manual*, 1998.

[For99] Formal Systems (Europe) Ltd. *FDR: User Manual and Tutorial, version 2.28*, 1999.

[Hoa85] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[IBM] Rational Rhapsody Architect for Systems Engineers. `http://www-142.ibm.com/software/products/us/en/ratirhaparchforsystengi`. Accessed: 11-04-2013.

[Mat] MathWorks. Stateflow and Stateflow Coder 7 User's Guide. `http://www.mathworks.com/products/stateflow`.

[MC12] A. Miyazawa and A. Cavalcanti. Refinement-oriented models of stateflow charts. *Science of Computer Programming*, 77(10):1151–1177, 2012.

[OMG10] OMG. OMG Systems Modeling Language (OMG SysML$^{\text{TM}}$). Technical report, 2010. OMG Document Number: formal/2010-06-02.

[SCJS10] Adnan Sherif, Ana Cavalcanti, He Jifeng, and Augusto Sampaio. A Process Algebraic Framework for Specification and Validation of Real-time Systems. *Formal Aspects of Computing*, 22:153–191, 2010.

[SCS06] T. Santos, A. L. C. Cavalcanti, and A. C. A. Sampaio. Object-Orientation in the UTP. In S. Dunne and B. Stoddart, editors, *UTP 2006: First International Symposium on Unifying Theories of Programming*, volume **4010** of *LNCS*, pages 20–38. Springer-Verlag, 2006.

[Spa] Sparx Systems' Enterprise Architect supports the Systems Modeling Language. `http://www.sparxsystems.com/products/mdg/tech/sysml/index.html`. Accessed: 11-04-2013.