Embedding Scenario-Based Modeling in Statecharts

Assaf Marron[1] and Yotam Hacohen[1] and
David Harel[1] and Andreas Mülder[2] and Axel Terfloth[2]

[1] Weizmann Institute of Science, Rehovot, Israel
[2] itemis AG, Lünen, Germany

Abstract. Scenario-based modeling (SBM) is an approach for creating executable models for reactive systems where each artifact specifies a separate aspect of overall system behavior. SBM has many advantages, including structural alignment with requirements, intuitiveness and incrementality, and it is available in visual languages (e.g., LSC), textual languages (e.g., Java, C++) and DSLs (e.g., SML). In this position paper, we argue that endowing the Statecharts visual formalism with SBM capabilities can significantly benefit software and system engineering, enhancing intuitive visualization of specifications and designs, and their scalability and amenability to formal verification. We demonstrate the position by amalgamating Statecharts and SBM within the YAKINDU Statechart Tools.

1 Introduction

Scenario-based modeling (SBM) [2, 8, 10], also termed *scenario-based programming* (SBP) and *behavioral programming* (BP), is an approach for creating executable models of reactive systems, such that each artifact (i.e., scenario) describes a separate aspect of the overall system behavior. Such scenarios can specify allowed, mandatory and forbidden behavior. New and refined requirements are often added incrementally in new artifacts, with little or no change to existing ones. The full system consists of the collection of these scenarios, taken together, either as is or transformed into code. SBM's advantages also include alignment of the code with the requirements, ease of understanding for a human reading the model or the code, the ability to directly execute the intuitive models, amenability to formal verification (see, e.g., [5]) and even succinctness of the formal representations as compared to alternatives (see, e.g., [6]). The syntax, semantics and an implementation of the approach were first introduced in [2, 8], with the graphical language of *live sequence charts* (LSC) and the Play-Engine tool. SBM was subsequently generalized and implemented in standard procedural languages, such as Java, C, C++ and JavaScript, in functional languages like Erlang, in the graphical block-oriented language Blockly and in textual modeling DSLs like SML/SCENARIOTOOLS.

An SBM scenario is often inter-object, involving behavior flow across multiple components. It thus may be contrasted with object-oriented or object-centric approaches for designing systems, based on each class's reactions to stimuli received from the environment and from other components.

A widely accepted visual formalism, which has been shown to be particularly suitable for object-oriented design (OOD), is Statecharts [4]. In various modeling environments there were elements that enabled integrating SBM with OOD and, in particular, with Statecharts. For example, in IBM's MDE tool Rational/Rhapsody, object behaviors can be modeled in the Statecharts while scenarios depicted in sequence diagrams can monitor that behavior. In [1], the Play-Engine and Rhapsody were actually integrated to exchange messages, and in [9] statecharts-based modeling was embedded and integrated within the PlayGo [7] LSC development environment, where LSCs and statecharts co-exist with fully defined composite semantics.

In this paper, we present a novel approach to integrating intra-object OO design with inter-object scenarios, by enhancing the syntax and semantics of the Statecharts formalism with SBM features. In Sec. 2, we present the main features of Statecharts and SBM and our research goals, namely, valuable SBM capabilities that we wish to propagate to Statecharts, and valuable Statecharts capabilities that were absent heretofore from existing SBM implementations. In Sec. 3, we describe a prototype implementation and its execution semantics as carried out by endowing the *YAKINDU Statechart Tools* of itemis corporation with SBM features. In Sec. 4, we illustrate the implementation via a small rocket-control simulator and game, and in Sec. 5, we conclude and discuss next steps. In the Appendices we extend the discussion of the example and the implementation in Yakindu, and lay the groundwork for the next research steps of this emerging language in outlining the key semantic and syntactic considerations.

2 Rationale for Amalgamating Statecharts and Scenarios

Statecharts were introduced in [4] as a visual formalism, in which complex behavior of reactive systems can be captured succinctly and intuitively. The key contributions over classical state machines were: *behavioral hierarchy*, represented as topological state containment; *orthogonality of concurrent behavior*, represented as side-by-side state machines; and *state awareness*, i.e., the ability to condition a transition upon another state machine being in a particular state. The Statetcharts formalism comes with formal semantics, and specifications are directly executable—enabling generation of prototype simulators and of the code of final system components. The `HumanPilot` region in Fig. 1 shows hierarchy and orthogonality[3]. Statecharts exist in leading development suites, including STATEMATE, Rhapsody [11], MATLAB/Simulink and SCADE and it is the formalism of choice for describing behavior dynamics in the OMG UML [13].

In SBM, each scenario describes an aspect of, or a requirement concerning, overall system behavior. The scenarios run in parallel and constantly present declarations of *requested events*, i.e., events that should be considered for triggering, *blocked events*, i.e., events whose triggering is forbidden, and events that the scenario neither requests nor blocks, but wishes to be notified when they occur. An infrastructure execution mechanism repeatedly synchronizes all scenarios, selects an event that is requested by some scenario and is not blocked by any scenario (termed an *enabled event*), and notifies all scenarios that requested that event or are waiting for it. These scenarios then resume execution, and can change their state accordingly, including declarations of requested, blocked and waited-for events. All scenarios are resynchronized and the process repeats. When no event is enabled, the system waits for an external environment event. Sensor scenarios or external code translate environment events and conditions into SBM events

[3] Please zoom-in on screen. The CR version will have larger font sizes, for printed copies.

that the model can react to, and actuator scenarios can translate SBM events into actual environment effects, by waiting for a given event (e.g., `MoveForward`) and calling an appropriate external environment API (e.g., a method that sets a motor object's speed).

The Statecharts-SBM amalgamation is motivated by research questions and engineering requirements that are part the general pursuit of making system development faster, cheaper, and with better quality: (a) In Statecharts, enable specification of exceptions and refinements to already-specified behavior in an incremental manner; i.e., add orthogonal state machines with little or no change to existing ones (SBM event blocking is a key contributor to this capability); (b) In the otherwise-flat SBM specifications, add hierarchy and containment, e.g. for organization and for managing context-dependent constraints (see also [3]); (c) In SBM, add the state-awareness ability that exists in Statecharts to explicitly condition a behavior upon the then-current state of another behavior; (d) Leverage in the amalgamated formalism the amenability of SBM to scalable compositional verification [5]. These goals come, of course, with challenges, like the risk of sacrificing the intuitive nature of a scenario when encoding it in statecharts.

3 Implementation and Semantics

Our implementation is based on the open-source YAKINDU Statecharts Tools (YSCT) [12]. The solution's key aspects are:

Hierarchical declaration of requested and blocked events. Each Statecharts state has now two additional properties (entered via the GUI editor): a set of requested events and a set of blocked events. The full sets of requested and blocked events when the system is at a given state s are the union of the respective sets in all the states containing s.

SBM-style event triggering. The Statecharts execution engine is modified as follows. A *superstep* begins when the statechart processes an external (environment) event or one that was explicitly raised within the statechart. The execution engine then chooses an enabled event — i.e., one that is requested by at least one active state and is not blocked by any active state — triggers it, and reacts to it using the normal Statecharts semantics based on transition labels. This process repeats until there are no more enabled events. The system then starts a new superstep by processing from its queue the next event that was triggered by the environment or explicitly raised by the statechart during the preceding superstep. If this queue is empty, the system waits for the next environment event.

4 An Example

We illustrate our implementation and semantics, using a small rocket-landing game/simulator (see Fig. 1): A rocket descends by gravity towards the ground or sea, aiming to land exactly on a landing pad. The user can choose between auto-pilot simulation or human-pilot control. A human pilot/player can press `right/left` buttons to shift the rocket accordingly in order to position it above the landing pad before it touches the surface. The entire landing area is curbed by two vertical walls. An adversary, e.g., ocean waves, arbitrarily shifts the landing pad, and the player has to re-adjust the rocket's position accordingly. In Auto-pilot mode an SBM-modelled controller issues the same `right/left` commands (for a related real-world control problem see, e.g., the SpaceX successful ocean landing at `https://www.youtube.com/watch?v=lEr9cPpuAx8`).

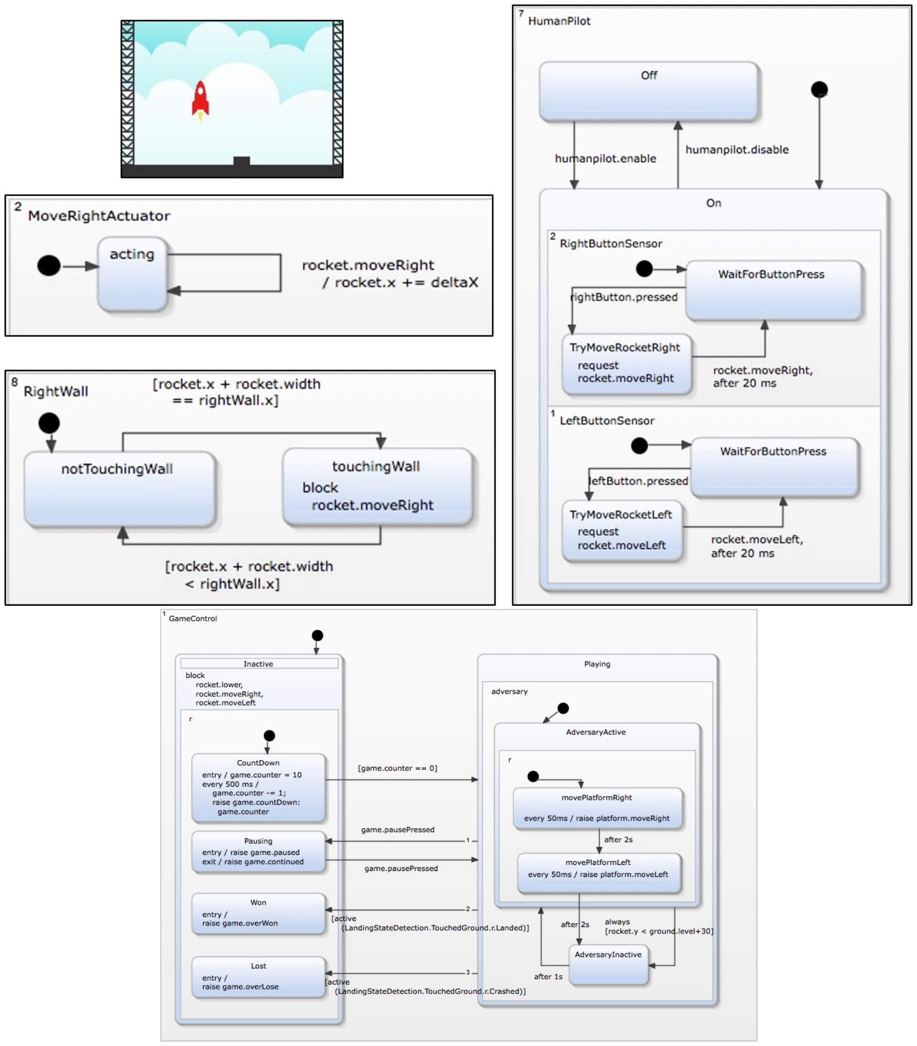
7 HumanPilot
Off
humanpilot.enable
humanpilot.disable
On
2 RightButtonSensor
WaitForButtonPress
rightButton.pressed
TryMoveRocketRight request rocket.moveRight
rocket.moveRight, after 20 ms
1 LeftButtonSensor
WaitForButtonPress
leftButton.pressed
TryMoveRocketLeft request rocket.moveLeft
rocket.moveLeft, after 20 ms
2 MoveRightActuator
acting
rocket.moveRight / rocket.x += deltaX
8 RightWall
[rocket.x + rocket.width == rightWall.x]
notTouchingWall
touchingWall block rocket.moveRight
[rocket.x + rocket.width < rightWall.x]
1 GameControl
Inactive
block rocket.lower, rocket.moveRight, rocket.moveLeft
r
CountDown
entry / game.counter = 10
every 500 ms / game.counter -= 1; raise game.countDown: game.counter
Pausing
entry / raise game.paused
exit / raise game.continued
Won
entry / raise game.overWon
Lost
entry / raise game.overLose
[game.counter == 0]
game.pausePressed
game.pausePressed
[active (LandingStateDetection.TouchedGround.r.Landed)]
[active (LandingStateDetection.TouchedGround.r.Crashed)]
Playing
adversary
AdversaryActive
r
movePlatformRight
every 50ms / raise platform.moveRight
after 2s
movePlatformLeft
every 50ms / raise platform.moveLeft
after 2s
always [rocket.y < ground.level+30]
AdversaryInactive
after 1s

Fig. 1: The rocket-landing game and a portion of its statechart.

The application's scenarios shown in Fig.1 are the orthogonal regions of player controls, movement actuators, the constraining walls' effects, win/lose/pause handling, and adversary behavior. The gravity force, autopilot behavior, landing detection and switching between human and auto-pilot modes, are shown in Appendix A.

Notice the introduction of SBM capabilities, where, e.g., human-pilot scenarios move the rocket unconditionally and they are composed with separate wall behavior that blocks this motion when needed. Likewise, when the rocket reaches the surface, all motion events are blocked. In addition, scenarios can be aware that their requests may not be granted, and withdraw them if they deem them to no longer be relevant (e.g., by labeling the transition exiting the requesting state also with a timeout or with an opposite move event). Here, this avoids an uncalled-for future rocket motion towards the wall, after moving away from the wall such that the event is no longer blocked.

Note also SBM's use of Statecharts features, like transitions based on Statecharts' state awareness, (e.g., from `Playing` to `Won` or `Lost`, based on the landing-detection state), and the hierarchical propagation of blocked events (e.g, in `Inactive`). A more detailed analysis of the SBM-Statecharts amalgamation appears in the appendix.

5 Discussion and Future Work

We have shown that merging scenarios and Statecharts is possible, and have argued that it is promising. In the Appendix, we analyze additional features, e.g., that transitions from a state that contains other states can be used to implement the classical `break` function (termed *cold violation* in SBM), and provide the basis for future enhancement of Statecharts with additional SBM-related features, like strict event ordering, and specification of liveness properties (SBM's *hot* and *cold*) for execution control and verification. We plan to also evaluate the amalgamation's benefits empirically, and to study its formal succinctness properties as compared to those of Statecharts and SBM alone.

References

1. D. Barak, D. Harel, and R. Marelly. Interplay: Horizontal scale-up and transition to design in scenario-based programming. *Lectures on Concurrency and Petri Nets*, pages 66–86, 2004.
2. W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *J. on Formal Methods in System Design*, 19(1):45–80, 2001.
3. A. Elyasaf, D. Harel, A. Marron, and G. Weiss. Towards synergistic integration of context-based and scenario-based development. *4th Workshop on Model-Driven Robot Software Engineering (MORSE; at STAF conference)*, 2017.
4. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
5. D. Harel, A. Kantor, G. Katz, A. Marron, L. Mizrahi, and G. Weiss. On composing and proving correctness of reactive behavior. *EMSOFT*, 2013.
6. D. Harel, G. Katz, R. Lampert, A. Marron, and G. Weiss. On the succinctness of idioms for concurrent programming. In *Proc. 26th Int. Conf. on Concurrency Theory (CONCUR)*, 2015.
7. D. Harel, S. Maoz, S. Szekely, and D. Barkan. PlayGo: towards a comprehensive tool for scenario based programming. In *ASE*, 2010.
8. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.
9. D. Harel, R. Marelly, A. Marron, and S. Szekely. Integrating inter-object scenarios with intra-object statecharts for developing reactive systems. 2017. Prelim. Rpt. (www.arXiv.org).
10. D. Harel, A. Marron, and G. Weiss. Behavioral programming. *Comm. of the ACM*, 55(7).
11. IBM Corporation. Rational rhapsody product family. URL: https://www.ibm.com/us-en/marketplace/rational-rhapsody (accessed 7/2018).
12. itemis Corp. Yakindu Statechart Tools site. http://www.statecharts.org/ (Accessed 7/2018).
13. OMG. *Unified Modeling Language Superstructure Specification, v2.0*. Aug. 2005.
14. W3C. State chart xml (scxml): State machine notation for control abstraction, 2015.

Appendices - Supplemental Material

Appendix A. Discussion of Demonstrated Statecharts and SBM Features

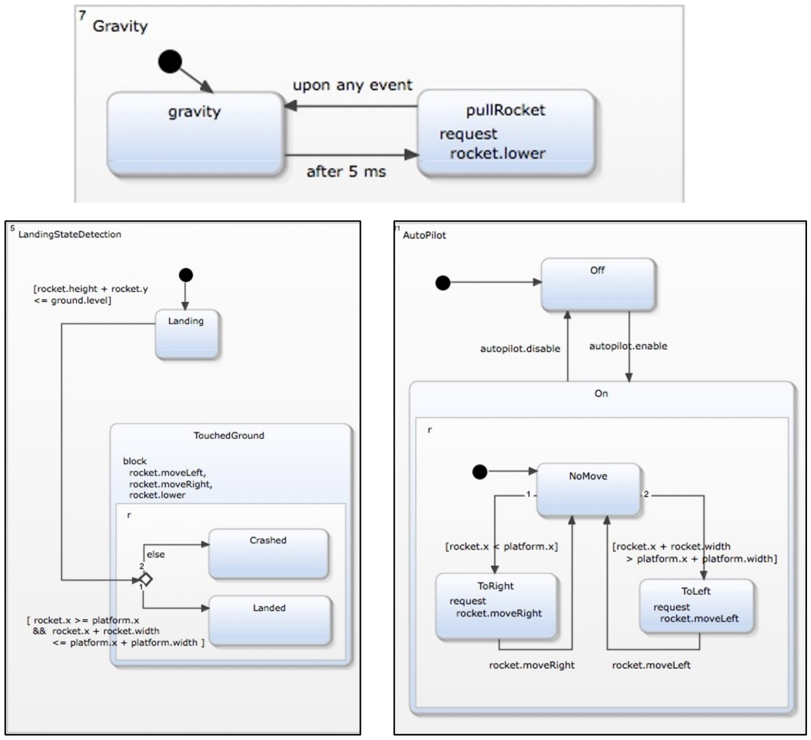

Fig. 2: Gravity, landing detection and auto-pilot statecharts.

Figs. 1, 2 and 3 depict the rocket-landing game Statechart almost in its entirety. Omitted are the similar handling of left movements and wall, and actuation of the gravity and adversary. We hope that the statechart diagrams are indeed self explanatory, as a model is expected to be, and add here some notes to draw attention to particular details and special properties of the SBM-Statecharts amalgamation.

Hidden components. The code that is not shown includes: the rendering of the game UI with rocket and landing pad location and movement, and the external sensor that listens to user controls and translates them into behavioral events.

Dynamic scenario instantiation. When the user activates and deactivates human pilot mode, the transition into the respective `humanPilot On` state respectively activates (i.e., instantiates), the scenarios that react to the player's control, illustrating one of the ways in which Statecharts can very intuitively introduce context-orientation into SBM (which SBM presently solves with dynamic object binding).

Incrementality. Aside from having developed this application incrementally, adding the autopilot and adversary behaviors with little or no change to existing scenarios, note

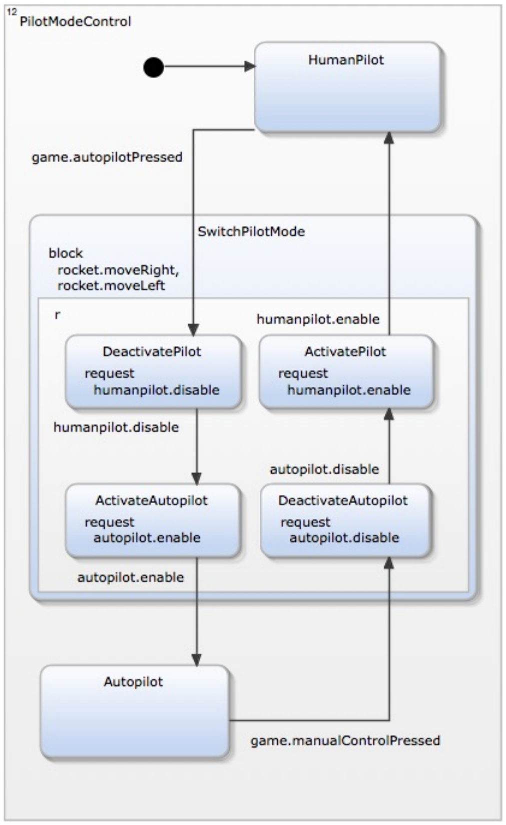

Fig. 3: Pilot mode switching statechart.

how new obstacles, e.g., rockets or mountains can be added incrementally, and how they can simply block the movement of the rocket upon additional conditions. Or, the rocket movement events could be requested by additional behaviors, say of engines or wind.

SBM's event unification in Statecharts. In SBM semantics, when an event that is requested by more than one scenario is triggered, all scenarios that requested it are resumed, and all observe this one occurrence of the event. If the intent is that the event will have separate occurrences per the requesting scenarios, the engineer must distinguish these events, e.g., by some parameter. This provides an added capability, compared with standard Statecharts semantics, where each execution of a `raise <event>` command results in a a separate, new, triggering of that event (note however that, some Statecharts implementations do support one triggering of an event per superstep).

State-awareness in SBM. The transition from the `Playing` to `GameEnded` is dependent not on internal events, but on awareness of whether other parts of the statecharts are in the states `crashed` or `landed`. In other SBM language in order for such status to be available to other components, it would have to be kept explicitly by the application in a dedicated variable, while in Statecharts it is readily available as a state. Clearly, Statecharts-like system-wide reflection by which a system component can discover the internal states of other components may undesirably break the otherwise-important encapsulation principle. However, with proper methodologies such reflection can enable system enhancements (perhaps 'patching') when, (a) one cannot modify existing code *and* (b) the referenced internal states indeed belong in the target component's interface.

Context Orientation in SBM. Note, e.g., the collective blocking of movement events during the scenarios contained in state/context `SwitchPilotMode`, or in all the states contained in the `Inactive` state which is reached when the game ends or is paused.

Explicit states in SBM. One of the attractive properties of SBM, say, the LSC language, is the ease with which one specifies sequential procedures like "S1: After events E1, E2, E3 occur in that order, do actions A1, A2, A3, A4, in that order". When written in Statecharts, the eight states of this executable scenario have to be specified explicitly. While this may seem excessive, this becomes valuable when considering reflection or strict ordering. E.g., at run time, another scenario S2 is able to inquire whether S1 has reached the actions segment or not, and react accordingly. The explicit states can also connect execution traces and formal verification results with program states that the engineer is already aware of, and that are not only derivatives of internal structures.

Intuitive `break`. In SBM, when a forbidden or interrupting event occurs in a *cold* state, the scenario (or a subchart) is simply exited. This is termed a *cold violation*, and is similar to a `break` statement in procedural programming. In other SBM languages the list of events that can cause a cold violation at a given state may not be obvious. With the SBM-Statecharts amalgamation this becomes clear, through the explicit and succinct labels of transitions exiting the given state or any state hierarchically containing it.

Retention of native Statecharts design choices. In the amalgamated solution engineers still have the choice of using pure object-oriented and Statecharts-specific features that may otherwise be 'against the grain' of SBM. E.g., in the rocket game, the auto pilot is driven by changes in object data and not by SBM events (though in LSC this is possible too). Another example are the transition arrows in pause-unpause or pilot-mode-switching scenarios that cross the boundaries of the containing state, where in SBM exiting arrows would begin at the perimeter of relevant context state, and transitioning into a specific internal behavior state would be specified by marking an initial states, and, when there is more than one, creating orthogonal behaviors. Developing design patterns and methodological recommendations for such choices are a future endeavor.

Appendix B. YAKINDU Implementation Specifics

This proptotype of an SBM extension to Statecharts is implemented using the YAKINDU Statechart Tools (YSCT) and consists of two parts: a language extension that supports the declaration or requested and blocked events and a runtime extension that processes SBM events. While YSCT has commercial components these extensions are all within YSCT's open-source core.

The Statechart Language Extension. Two statement types, *block* and *request*, were added to the textual behavioral description of states extending YSCT's standard statechart language definitions (see the YSCT language documentation in `www.itemis.com`). This language consists of a set of modular meta-models and syntax definitions that completely cover the graphical and textual parts. Like any textual notation in YSCT the additional statements are defined using a meta model fragment that specifies the abstract syntax and a set of grammar rules that define the concrete textual syntax. Both parts are pure extensions to the underlying YSCT meta models and grammar definitions, hence all statechart language concepts are still available also in combination with SBM.

The Statecharts Runtime Extension. The runtime extensions are implemented in YSCT Java code generator (Yakindu also supports C and C++). The code generator relies on the new language concepts to calculate requested and blocked event sets and implements the event triggering mechanism according to the defined semantics (see Sec. 3). A comparable addition also to YSCT's model interpreter infrastructure, which is used for direct simulation of statecharts, is deferred for future work.

SBM Statechart Domain. YAKINDU Statechart Tools support the concept of *Statechart Domains* which are pluggable extensions that allow the definition of domain-specific statecharts and statechart dialects. Statechart domains may contribute extensions and substitutes to various parts of the statechart language, type system and type inferrer, validation rules, execution semantics, interpreter, and code generators. The current extensions are implemented in a separate statechart domain.

Internal vs. External Extension. The current implementation addresses statechart internal concepts and concepts that are part of the statecharts external interface are not covered. This will become necessary as soon as the implementation has to support SBM concepts across multiple statecharts. Then, the requested and blocked event sets and state information for state-awareness must be added to the statecharts' public interfaces and to the new event triggering mechanism which which might have to be implemented outside of the statecharts. Another consideration in this case would be adopting for YSCT the distributed SBM execution semantics.

Appendix C. Next-steps blueprints

In this section we discuss the various considerations that affect the syntactic and semantics decisions to be taken when further extending the amalgamation of Statecharts and SBM in YAKINDU as well as in other platforms.

Must vs. May. In some SBM languages (e.g., LSC) scenario states can be tagged as *hot* or *cold* meaning where if scenario that is in a hot state must eventually transition into a cold one. If in this state the only declaration is that of one requested event, then this means that this event must happen (In the PlayGo LSC tool, the hotness/coldness of states is derived from the hot/cold annotation of the requested and waited for events). States being hot or cold affects both run-time event selection and the determination of whether a run complies with the (liveness properties of the) specifications. While adding hot/cold annotation to states and/or transitions in YAKINDU would be straightforward, the semantics is more delicate. One possibility is to add hot/cold annotation only to states (an not to transitions) and prioritize selecting enabled events that exit a hot state over ones that exit a cold state.

Idioms for dealing with time. In Statecharts and in LSCs and some BP implementations, time is a separate entity with its own idioms. In addition, user-written code can create events indicating periodic time ticks or the passing of desired time durations. It is yet to be determined what features are really necessary for development productivity and for verifiability of the models.

Waiting for a state's own requested events. In most SBM implementations, when a scenario requests an event, it is notified when the event is triggered. In the SBM-YAKINDU amalgamation, we captured requested events as a state property, creating some redundancy with the transition labels, which include all events that the scenario should be notified about in that state. The redundancy may be eliminated by noting the requested events only on the respective transitions, and having distinct notation for events that are requested (and waited for) and for those that are only waited for.

Strict event ordering. In LSC and SML, one can annotate a scenario as strict meaning: (a) if an event mentioned in this scenario is triggered out of order (for this scenario) as driven by the environment or by another scenario, this scenario exits; (b) if a scenario is in a hot state then all events mentioned in the scenario which are not enabled are considered forbidden and cannot be triggered by other scenarios, and if such an event is triggered by the environment (as this cannot be blocked), a hot violation occurs. Note that events that are not specified in the scenario can occur at any time, and the scenario is oblivious to them. A similar function can be readily added to the Statecharts execution semantics, however, one needs to carefully determine the scope of the strictness, e.g., how to treat a request by a containing state, when the contained behavior is in a hot state.

Automatic generation of Statecharts from Natural Language (NL). The existing NL LSC interface that formalizes NL requirements (like "when the driver presses the brake pedal, the car slows down") as executable LSCs can be adjusted for also creating Statecharts. Note that, additionally, one can describe multiple transitions in the same statechart in multiple sentences of the form "when the car is in State `Driving`, and event `pressPedal` occurs, then the car transitions into state `slowingDown`". The NL specification may be tied to existing textual descriptions of statecharts like SCXML [14] together giving engineers a choice of both visual and textual specifications.

Multi-statechart support. The rocket-landing example one statechart, divided into regions. Applications that use multiple statecharts can presently be developed in YAKINDU by adding application-specific glue code. It is our plan to add to YAKINDU's GUI editor and execution visualizer native support for multiple Statecharts (similar to what was done in the PlayGo tool in in [9]).

Distributed and dynamic environments. We plan to add demonstration multiple concurrent flows that are instantiated dynamically and are associated with a dynamic object model where objects appear and disappear, and the events carry additional data. Such capabilities were already shown in SBM natively in BP in PlayGo and SML as well as in [9] in the tight-coupling SBM and Statecharts, hence it should be straightforward to extend it to the amalgamated solution.