Formal Compositional Semantics for Yakindu Statecharts

Bence Graics, Vince Molnár
Budapest University of Technology and Economics,
Department of Measurement and Information Systems
Budapest, Hungary
Email: bence.graics@inf.mit.bme.hu, molnarv@mit.bme.hu

***Abstract*—Many of today's safety-critical systems are reactive, embedded systems. Their internal behavior is usually represented by state-based models. Furthermore, as the tasks carried out by such systems are getting more and more complex, there is a strong need for compositional modeling languages. Such modeling formalisms start from the component-level and use composition to build the system-level model as a collection of simple modules. There are a number of solutions supporting the model-based development of safety-critical embedded systems. One of the popular open-source tools is Yakindu, a statechart editor with a rich language and code generation capabilities. However, Yakindu so far lacks support for compositional modeling. This paper proposes a formal compositional language tailored to the semantics of Yakindu statecharts. We propose precise semantics for the composition to facilitate formal analysis and precise code generation. Based on the formal basis laid out here, we plan to build a complete tool-chain for the design and verification of component-based reactive systems.**

I. Introduction

Statechart [1] is a widely used formalism to design complex and hierarchical reactive systems. Among the many statechart tools, our work is based on the open-source Yakindu[1], which supports the development of complex hierarchical statecharts with a graphical editor, validation and simulation features. Yakindu also supports source code-generation from statecharts to various languages (Java, C, C++).

The requirements embedded systems have to meet are getting more and more complex. Therefore, the models created for such systems tend to become unmanageably large, which encumbers extensibility and maintenance. Instead, the resulting models could be created by composing smaller units. These units interact with each other using the specified connections, thus implementing the original behavior. There are several tools that aim to support this methodology.

SysML [2], [3] tools have a large set of modeling elements which enables their users to express their thoughts and ideas as freely and concisely as possible. On the other hand, they rarely define precise semantics, which encumbers code generation and analysis. BIP [4]–[6] is a compositional tool with well-defined semantics that supports the formal verification of modeled systems. Source code generation is also possible with this tool. Scade[2] [7], [8] is a tool that unifies the advantages of design and analysis tools. It supports the generation of source code as well as the formal verification of the modeled system. It is a commercial tool and does not support extensibility. Matlab Stateflow [9] is an environment for modeling and simulating decision logic using statecharts and flow charts. It is a leading tool for composing state-based models in the domain of safety-critical embedded systems. It supports the encapsulation of state-based logics which can be reused throughout different models and diagrams.

Unfortunately, Yakindu does not support composition features. The main goal of our work is to create a tool that enables the users to compose individual statechart components into a single composite system by constructing connections through ports. The ultimate goal of this work is to enable code generation and formal verification of composite models with model transformations based on the proposed semantics.

We will call this type of composition an event-based automata network, as opposed to dataflow networks, which can be considered message-based automata networks in this sense. In event-based automata networks, data is only of secondary importance – the occurrence of the event is in focus. In message-based settings, data is more significant, thus message queues are desirable to buffer the simultaneous messages.

This paper is structured as follows. Section II presents the semantics of Yakindu statecharts serving as the basis of the compositional language. The syntax and semantics of the compositional language along with an example are introduced in Section III. Finally, Section IV provides concluding remarks and ideas for future work.

II. Abstracting Yakindu Statecharts

Yakindu adopts a statechart formalism which is the extension of the well-known state machine formalism. Statecharts support the definition of auxiliary variables as well as concurrency and state refinement. This section introduces a syntactical abstraction of Yakindu, i.e. the actual model elements are ignored. We deal only with the input and output events in addition to the actual state configuration, but not the semantics. This way we can generalize the composition

[1]https://itemis.com/en/yakindu/statechart-tools/

[2]http://www.esterel-technologies.com/products/scade-suite/

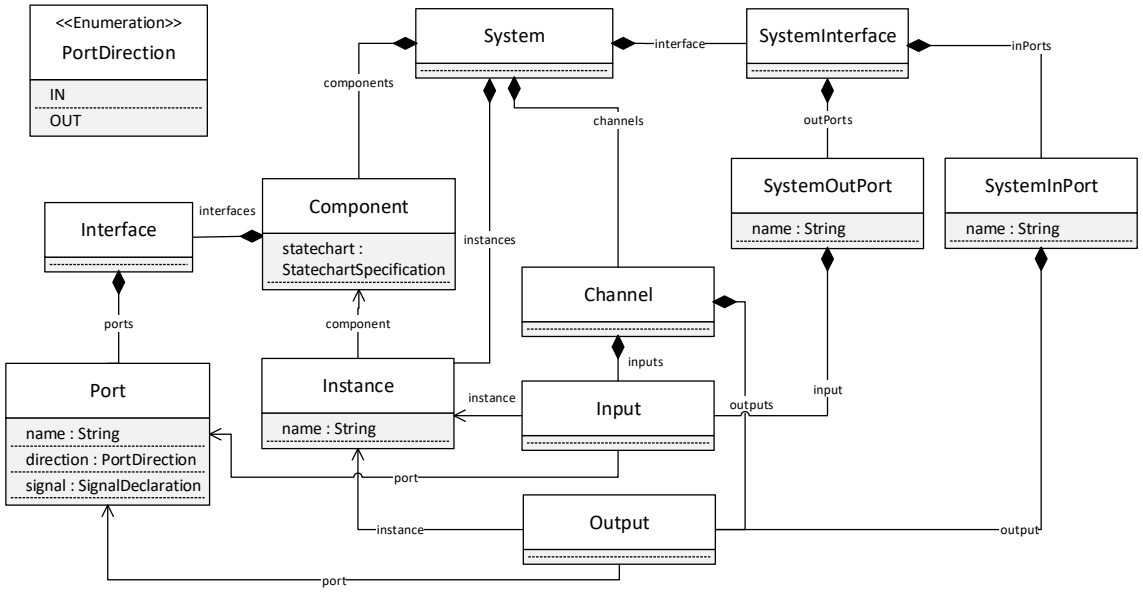

Fig. 1. Metamodel of the compositional language.

of abstract models with minimal restrictions to the usable formalisms. In this approach, a Yakindu statechart is considered a 5-tuple: $\mathbb{S} = \langle I, O, S, s_0, T \rangle$ where:

- I is a finite set of input events (from the environment)
- O is a finite set of output events (for the environment)
- $S = \{s_1, s_2, \cdots, s_n\}$ is a finite set of states, including a state configuration and values of variables
- $s_0 \in S$ is the initial state
- $T \subseteq (2^I \times S) \times (S \times 2^O)$ is a finite set of transitions, that represent changes of state in response to a set of input events and generate a set of output events

Yakindu statecharts adopt a turn-based semantics. The following paragraphs introduce the interpretation of *turns* as well as how the raising of events is associated to them.

Events represent signal receptions. There are two types of events: *simple* or *void* events and *parameterized* or *typed* events. The latter enables the modeling of parameterized signals, which can describe additional details. Note that multiple input events can be raised in a single turn according to the abstract formalism defined above. In Yakindu the raising of events is interpreted as setting a boolean flag to true. Yakindu therefore does not support message queues. Owing to this semantics raising the same *simple* event in a particular turn once or several times has the same effect. On the other hand, *parameterized* events are defined by their parameter as well, so a new event raising with a different parameter overwrites the former one. Although this behavior is an essential part of the semantics of Yakindu, it is not relevant either in the abstract formalism presented above or the semantics of the composition language defined in Section III-B.

All turns consist of two distinct sections, a *raising* section and a *running* section. In the raising section input events of the statechart are raised as presented in the previous paragraph. This is followed by the running section where a new stable state of the statechart is defined. It starts with the examination of the transitions going out of the particular state configuration. The goal is to specify the firing transition. At this point a *race condition* might exist if multiple outgoing transitions are enabled, e.g. more of them are triggered by raised input events. Yakindu intends to solve ambiguity by introducing the concept of *transition priority*: users can specify which of the outgoing transitions of a state should be fired in case of a race condition by defining a total ordering of the transitions. The firing transition specifies the next stable state of the statechart, including the state configuration, values of variables and events for the environment.

III. Language for Composition

This section defines the syntax of the compositional language and introduces the semantics the composite system conforms to. This semantics is heavily influenced by the statechart semantics defined by Yakindu and strives to address some of its problems, e.g. gives the ability to parallel regions to communicate with each other.

A. Syntax

Figure 1 depicts the metamodel of the compositional language. The root element in the metamodel is the *System*. A *System* contains *Components* which refer to Yakindu statecharts as well as *Instances* of such *Components*. Each *Component* has an *Interface* that contains *Ports*. Through *Ports*, signals of statecharts can be transmitted or received according to their directions.

Channels can be used for defining the emergent behavior of the composite system. A *Channel* has one or more *Inputs* and one or more *Outputs*. An Input of a *Channel* connects to an output *Port* of an *Instance* and vice versa. Whenever a

Channel receives a signal through any of its *Inputs*, the signal is sent to each *Output*, i.e. to the corresponding input *Ports* of *Instances*. The language does not support connecting *Ports* of the same direction and a validation rule is defined that marks incorrect connections.

The language supports the definition of an interface through which the composite system interacts with its environment. This is the *SystemInterface* that contains *SystemPorts*. *SystemPorts* are aliases of *Ports* of *Instances*. If a signal arrives to a *SystemInPort*, it will be forwarded to the *Port* of the referred *Instance* instantly. *SystemOutPorts* work similarly, but with output *Ports* of *Instances*.

For ease of understanding, an example is presented that defines a composition of statecharts using the specified compositional language. The system consists of two *Components*, *CoffeMachineComponent* and *LightComponent* referring to a coffee machine (CoffeMachine) statechart and a light switch (LightSwitch) statechart, respectively. CoffeMachine has signal declarations for turning it on and off, for ordering a cappuchino and for putting its light on and off. A LightSwitch models a lamp that can be turned on and off.

```
// System interface definition
interface {
    in {
        on : machine.on
        off : machine.off
        cappuchino : machine.cappuchino
    }
}

// Component interface definitions
CoffeeMachine CoffeMachineComponent {
    interface {
        on : IN on
        off : IN off
        cappuchino : IN cappuchino
        lightOn : OUT flashLight
        lightOff : OUT turnOffLight
    }
}

LightSwitch LightComponent {
    interface {
        on : IN onButton
        off : IN offButton
    }
}

// Component instantiations
CoffeMachineComponent machine
LightComponent light

// Channel definitions
channels {
    [machine.lightOn] -> [light.on]
    [machine.lightOff] -> [light.off]
}
```

Note that a composite system description constists of the following parts:

- System interface definition: All input *Ports* of *machine* are published to the interface of the system enabling the users to turn *machine* on and off or order a cappuchino.
- Component interface definitions: *CoffeMachineComponent* refers to *on*, *off* and *cappuchino* through input *Ports* (denoted by the IN keyword) and *flashLight*, *turnOffLight* through output *Ports* (denoted by the OUT keyword). Both signal declarations of LightSwitch are referred to by input *Ports*.
- Component instantiations: Both *Components* are instantiated: *machine* and *light*.
- Channel definitions: The output *Ports* of *machine* are connected to the input *Ports* of *light*, making it possible for *machine* to turn on *light* at choice.

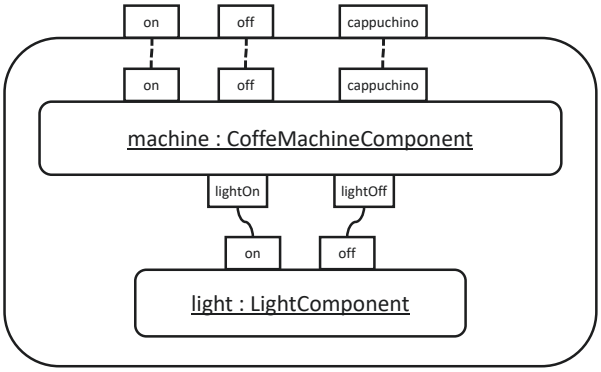

Fig. 2. A composite system of a CoffeMachine and a LightSwitch statechart.

Figure 2 depicts the composite system described by the previous code section. Note that the individual components of the system are encapsulated. Interactions can be specified only through the defined interface.

B. Semantics

During the design of the semantics one of our goal was to define a language that enables the reuse of the source code generator of Yakindu. Therefore the semantics of supported Yakindu statecharts elements had to be considered, most importantly event raising.

This section introduces the semantics of the compositional language. The compositional language enables to create a *composite system*, that is formally a 4-tuple: $C = \langle SC, CA, IN, OUT \rangle$ where:

- $SC = \{\langle S_1, s_1^0, T_1, I_1, O_1\rangle, \cdots, \langle S_n, s_n^0, T_n, I_n, O_n\rangle\}$ is a finite set of state machines.
- $I = \bigsqcup_{j=1}^{n} I_j$, i.e. the union of all in events of state machine components
- $O = \bigsqcup_{j=1}^{n} O_j$, i.e. the union of all out events of state machine components
- $CA \subseteq 2^O \times 2^I$, i.e. channel associations relate a finite set of outputs to a finite set of inputs
- $IN \subseteq I$, i.e. the input interface is a subset of the union of the in events of state machine components
- $OUT \subseteq O$, i.e. the output interface is a subset of the union of the out events of state machine components

A sequence of steps $\varrho = (\tau_1, \tau_2, \cdots)$ is called a *complete run* of C if the following conditions hold.

- $\tau_j = (\underline{s}_j, i_j, \underline{s}'_j, \underline{o}_j)$ is a single step that consists of a state vector representing each state of each component before the step, a finite set of inputs, a state vector representing each state of each component after the step and a finite set of outputs generated by each state machine components, where for all $1 \leq k \leq n$ at least one of the following conditions holds:
  - $(i_j \cap I_k, \underline{s}_j[k], \underline{s}'_j[k], \underline{o}_j[k]) \in T_k$, i.e. if a transition is defined in a state machine component that is triggered by the input set, then the transition fires taking the state machine to its target state and producing the corresponding outputs;
  - $(\underline{s}_j[k] = \underline{s}'_j[k] \wedge \underline{o}_j[k] = \emptyset \wedge \nexists s', o' : (i_j \cap I_k, \underline{s}_j[k], s', o') \in T_j)$, i.e. a component is allowed to do nothing if and only if it has no transition that is triggered by input i_j in state $\underline{s}_j[k]$;
- $\underline{s}_1 = (s_1^0, s_2^0, \cdots, s_n^0,)$, i.e. at the beginning of the run, all state machine components are in their initial states;
- $\underline{s}'_j = \underline{s}_{j+1}$, i.e. the state vector at the end of a step and at the beginning of the next step are equal;
- $tgd(\bigcup_{k=1}^{n} \underline{o}_j[k]) \subseteq i_{j+1} \subseteq tgd(\bigcup_{k=1}^{n} \underline{o}_j[k]) \cup IN$ where $tgd(\Omega) = \bigcup_{\omega \in 2^{\Omega}} \omega \circ CA$, i.e. the inputs of a step is at least the inputs triggered through a channel by outputs of the previous step and maybe some additional events of the input interface;
- ϱ is either infinite or the following condition holds:
  - $\nexists (o, i) \in CA \colon o \cap o_n \neq \emptyset$, i.e. the execution of steps can terminate only if the last step does not produce any outputs that will be inputs in the next step.

A partial run of a composite system can be any prefix of a complete run (any other sequence is not considered to be a behavior of the composite system).

It is important to note that message queues (buffering) are not included, the semantics guarantees only that event raising and event receptions are in a causal relationship. Therefore, if a component does not buffer events (such as Yakindu), parameterized events may overwrite each other.

The operational semantics presented above provides a way to reduce the semantics of the composite system to the semantics of the components. To formally analyze the system, denotational semantics has to be provided, e.g. by model transformations converting the composite system model into a formal model, in accordance with the operational semantics.

IV. Conclusions and Future Work

Yakindu is a popular open-source tool for the design of statechart models with support for code generation. It has a rich language to model a single hierarchical statechart, but it lacks the ability to compose statecharts into a component-based model. For the design of complex, embedded reactive systems, compositionality is essential to handle the design complexity. Moreover, a precise formal semantics is necessary to facilitate code generation and formal analysis.

The defined compositional language enables to instantiate Yakindu statecharts, specify ports for these instances and join these instances through port connections. The semantics of the language is well-defined and suits the statechart semantics of Yakindu soundly.

Subject to future work, we plan to extend the compositional language to allow hierarchical compositions, i.e. the composition of composite systems. Additionally, we intend to design a whole framework around the language that *1)* enables the generation of source code which connects the Yakindu statecharts according to the defined semantics and *2)* provides automated model transformation to formal models of composite systems on which exhaustive analysis can be performed by model-checkers.

The automatic model transformers will utilize a graph-pattern-based approach to generate the traceability information that will facilitate the back-annotation of the results of formal analysis to the engineering domain. This way, we hope to support formal verification without requiring the designers to get familiar with the formal languages involved.

Acknowledgment

This work was partially supported by IncQuery Labs Ltd. and MTA-BME Lendület Research Group on Cyber-Physical Systems.

References

[1] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, Jun. 1987. [Online]. Available: http://dx.doi.org/10.1016/0167-6423(87)90035-9

[2] OMG, *OMG Systems Modeling Language (OMG SysML), Version 1.3*, Object Management Group Std., 2012. [Online]. Available: http://www.omg.org/spec/SysML/1.3/

[3] L. Delligatti, *SysML Distilled: A Brief Guide to the Systems Modeling Language*, 1st ed. Addison-Wesley Professional, 2013.

[4] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in BIP," in *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, ser. SEFM '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 3–12. [Online]. Available: http://dx.doi.org/10.1109/SEFM.2006.27

[5] I. Konnov, T. Kotek, Q. Wang, H. Veith, S. Bliudze, and J. Sifakis, "Parameterized Systems in BIP: Design and Model Checking," in *27th International Conference on Concurrency Theory (CONCUR 2016)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), J. Desharnais and R. Jagadeesan, Eds., vol. 59. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 30:1–30:16. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2016/6167

[6] M. D. Bozga, V. Sfyrla, and J. Sifakis, "Modeling synchronous systems in BIP," in *Proceedings of the Seventh ACM International Conference on Embedded Software*, ser. EMSOFT '09. New York, NY, USA: ACM, 2009, pp. 77–86. [Online]. Available: http://doi.acm.org/10.1145/1629335.1629347

[7] H. Basold, M. Huhn, H. Günther, and S. Milius, "An open alternative for SMT-based verification of SCADE models," in *Proc. 19th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'14)*, ser. Lecture Notes Comput. Sci., F. Lang and F. Flammini, Eds., vol. 8718. Springer, 2014, pp. 124–139.

[8] R. Venky, S. Ulka, A. Kulkarni, and P. Bokil, "Statemate to scade model translation," in *ISEC '08: Proceedings of the 1st conference on India software engineering conference*. New York, NY, USA: ACM, 2008, pp. 145–146. [Online]. Available: http://dx.doi.org/10.1145/1342211.1342245

[9] J. Chen, T. R. Dean, and M. H. Alalfi, "Clone detection in matlab stateflow models," *Software Quality Journal*, vol. 24, no. 4, pp. 917–946, 2016. [Online]. Available: http://dx.doi.org/10.1007/s11219-015-9296-0