Mix-and-Match Composition in the Gamma Framework

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

***Abstract*—The Gamma Statechart Composition Framework is a modeling tool that supports the hierarchical composition of statechart components with well-defined compositonal semantics, as well as source code generation and formal verification. The purpose of the framework is to provide common ground for modeling and verification tools, as well as to support component-based system design building on existing statechart modeling tools. Currently, the framework has a single composition semantics, which executes the components in a lockstep fashion. This paper presents a new composition language for the Gamma Framework, adding support for two more semantics. Asynchronous-reactive semantics supports the proper abstraction of distributed communication, synchronous-reactive supports the modeling of highly synchronous communication, and cascade composition is a sequential decomposition of a single function.**

I. Introduction

Statecharts [1] are a widely used formalism to describe the behavior of reactive systems, which process stimuli from the environment and react with respect to their internal states. Statecharts introduce complex elements to aid the modeling of such systems, e.g., variables, hierarchical state refinement, history states and complex transitions (e.g., inner transitions).

The requirements such systems have to meet are getting more complex, which can result in very large system models, hindering verifiability, maintenance and extensibility. A well-known solution for managing complexity is decomposition. In case of statecharts, one way of decomposition is to define individual reactive components that, by means of communication, realize a more complex behavior. There are a number of modeling tools that aim to support this practice with various model-driven software development techniques such as code generation and verification.

The Gamma Statechart Composition Framework is one such tool, providing a layer for composing individual statechart components (possibly coming from other tools) while extending the capabilities of automatic code generation and verification and validation (V&V). Functionalities of the Gamma Framework as well as a case study are presented in [2]. In this paper, we propose a new composition language for Gamma that enables the hierarchical mixing of different composition semantics, with a focus on features and modeling aspects.

Asynchronous-reactive: Such models represent a set of components that are executed independently. Asynchronous components communicate by means of messages and message queues. This semantics is convenient when decomposition is both logical and physical, e.g., for distributed controllers.

Synchronous-reactive: A synchronous model represents a coherent unit consisting of strongly coupled but concurrent components which communicate in a synchronous manner using signals. This semantics is suitable for the logical decomposition of synchronous systems or the modeling of hardware-related designs.

Cascade: Cascade models represent a set of filters that are applied sequentially to derive an output from an input. This variant supports the design of adapters, runtime monitors and units with a batch-like execution.

The rest of the paper is structured as follows. Section II presents a short summary of tools that inspired the design of the composition language. The elements of the language itself and an example are introduced in Section III. Finally, Section IV provides concluding remarks and ideas for future work.

II. Related Tools

We present three tools that also support the *mixing of different composition semantics* for component-based systems.

Ptolemy II[1] [3], [4] is an open-source framework aiming to support the modeling of hierarchical composite systems with numerous component variants and communication semantics. Communication semantics are determined by *directors*, which are responsible for defining a *model of computation* on a particular hierarchy level. Various directors (e.g., process network, synchronous data flow or synchronous reactive) can be combined through different hierarchy levels, facilitating the design of complex model behavior. One of the main strengths of Ptolemy II is its simulation capability. However, source code generation and formal verification are not supported.

BIP[2] [5], [6] is a modeling framework that focuses on the formal definition of heterogeneous systems. BIP offers a language to define hierarchical composite models, where the interactions of constituent components are based on *synchronization*. BIP defines a clear operational semantics that describes the behavior for both atomic components (transition

[1] http://ptolemy.eecs.berkeley.edu/

[2] http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html?lang=en

system model) and compound components (rigorous rules for component interactions). Moreover, BIP offers a comprehensive tool set, which provides model transformers for third-party models, code generators, and formal verification capabilities.

Stateflow[3] [7] is a commercial framework that supports the modeling of reactive systems. Stateflow supports the design of statecharts and provides various scheduling algorithms. Furthermore, Stateflow supports the simulation, validation and verification of models as well as source code generation. Stateflow is a mature commercial software product with professional support. As such, the possibilities of extending or integrating it with other software is limited. Furthermore, it is very expensive even for research purposes.

Gamma was designed to provide an extensible framework for the design of statechart-based composite systems, inspired by the merits and limitations of the presented tools.

III. The Gamma Composition Language

The *Gamma Composition Language* (GCL) supports the definition of communicating composite models built from individual reactive components. The design of composite systems starts with the definition of interfaces, which define the possible event types that can be transmitted between components. The interfaces can be realized by ports of components, which can be connected with channels, enabling communication. Communicating components can be wrapped by composite models, creating an independent composite reactive unit with rigorously defined interaction patterns.

A. Communication Elements

In the GCL, components communicate through *ports*. Each port defines a point of service through which certain *event notifications* can be sent or received. An event notification (or event for short) is a piece of information passed between components, which can also have *parameters* to forward data. An event is called *message* in case of asynchronous components and *signal* in case of synchronous components. Events are declared on *interfaces*, which may be realized by ports. An event may be declared as *input*, *output* or *in/out*, which means that it can be received, sent or both through the realizing port. The declared direction will be reversed, however, if the port does not *provide*, but *require* the interface, which are the two possible modes in which a port can realize an interface. A *broadcast interface* is a special type of interface on which every event is *output*. This approach is similar to how the Franca Interface Definition Language defines interfaces.[4]

The concept of input and output events with the two modes of interface realization may be unusual at first sight, since ports in UML-like modeling languages support event reception and method declarations only, both of which are services that can be invoked. Our goal with this solution is to investigate the possibilities of precise interface-based communication in the domain of reactive systems. On the other hand, it is possible to use only *out* events on every interface – then provided mode is "output" mode and required mode is "input" mode.

The following snippet defines an interface that has a single input event with an integer parameter as well as an interface with a single output event, that is, a broadcast interface.

```
interface Status {
  in event query
  // Event with integer parameter
  out event status(code : integer)
}
interface PoliceInterrupt { // A broadcast interface
  out event interrupt
}
```

B. Components

Components are the basic building blocks of composite Gamma models. The declaration of a component always defines one or more ports in the header which may be referred to in the definition, as presented in the following snippet.

```
sync Crossroad [
  port police : requires PoliceInterrupt,
  port priorityLight : provides LightCommands,
  port secondaryLight : provides LightCommands
]
```

A component can be either *atomic*, wrapping a single statechart, or *composite*, wrapping one or more component instances. Statecharts can be defined in the *Gamma Statechart Language* (GSL), presented in [2]. Composite components may adopt three types of semantics, but regardless of that, they will comprise of the same basic modeling elements: component instances, port bindings and channels.

a) Component instance: Component declarations can be instantiated in composite component definitions, but they may not contain themselves (not even transitively). Composite components may contain instances of other composite components, yielding a hierarchical composition. Component instances inherit the declared ports through which they can communicate.

```
// Instance of a component type
component crossroadInstance : Crossroad
```

b) Port binding: The port binding element is responsible for mapping the declared ports of a composite component to one of the ports of its constituent components.

```
// Binding component ports to internal ports
bind police -> controller.policeInterrupt
bind priorityLight -> priorityLight.lightCommands
```

In the example above, the *police* port of the composite Crossroad component declaration is mapped to the *policeInterrupt* port of the component instance *controller*. This means that events received through the *police* port will be instantaneously forwarded to the *policeInterrupt* port, and events sent through the *policeInterrupt* port will be sent through the *police* port.

c) Channel: Communication may happen through *channels*. *Simple channels* can connect two ports if they implement the same interface but in different modes, i.e., the signal directions will be exactly the opposite on the two ports. *Broadcast*

[3]https://www.mathworks.com/products/stateflow.html
[4]http://www.eclipse.org/proposals/modeling.franca/

channels are similar, but they allow a single port *providing a broadcast interface* to be connected to multiple ports *requiring the same broadcast interface*. Note that the language will restrict the usage of channels in certain composition semantics, which is discussed in Section III-C.

```
// A simple and a broadcast channel
channel [controller.priotityControl] -o)-
  [priorityLight.control]
channel [controller.policeInterrupt] -o)-
  [priorityLight.police, secondaryLight.police]
```

C. Composite Component Variations

Composite components can be *synchronous* or *asynchronous*, which will determine how they receive events and how they execute their constituent components.

1) Synchronous Components: Synchronous components represent models that communicate in a synchronous manner using *signals*. Constituent components of synchronous composite components must be synchronous themselves and are executed in a lockstep fashion, triggered by the enclosing asynchronous component or an external actor. When executed, synchronous components process incoming signals and produce output signals in accordance with their internal states. Input signals are not queued but sampled: upon execution, the component can access the most recent signal for every event on every port since the last execution (if any). Similarly, output signals are reset in every execution and every output event on every port may get a new signal assigned to it. Synchronous components in Gamma are *synchronous* and *cascade composite components* that can be freely mixed and *statechart definitions* as atomic components.

a) Synchronous composite component: During the execution of a synchronous composite component, every constituent component is executed exactly once. The execution has two phases: *1)* all components sample their inputs, then *2)* outputs and new internal states are computed. This strategy guarantees that an output generated by a component will affect other components only in the next execution – therefore the execution order is insignificant and the composition of deterministic components yields a deterministic composite component. This behavior was one of the most important design goals for synchronous-reactive compositions.

The only case that could violate the deterministic behavior would arise when an input event has more than one sources. In this case, one signal would overwrite the other, and their "order" would be unspecified. To avoid this, the language restricts the way ports can be connected with channels: every port may be the endpoint of exactly one channel *or* be bound to exactly one port of the enclosing component.

b) Cascade composite component: Conceptually, cascade composite components represent a set of "filters" through which inputs are transformed into outputs. Therefore, constituent components will immediately see the output signals of other components in the same composite component. This is achieved by merging the sampling and computation phases and performing them both when executing a component. Deterministic behavior is achieved by executing the components in the *order of their instantiation*.

This strategy guarantees that the effects of an input signal will be observable in the immediate output of the composite component (in case of synchronous composite components, the effect may be delayed by several execution cycles). Signals sent through feedback connections (i.e., when a component sends a signal to another that comes earlier in the execution order) will be delayed until the next execution (they may be used for e.g., abort signals from monitor components). Note that the synchronous and cascade composite components are semantically incompatible, i.e., there are modeling designs which can be described in only one of the variants.

The typical arrangement of a synchronous or cascade composite component definition is as follows.

```
[sync|cascade] Crossroad [
  // Port declarations
  port police : requires PoliceInterrupt
  ...
] {
  // Component instances
  component controller : Controller
  ...
  // Binding composite model ports to internal ports
  bind police -> controller.policeInterrupt
  ...
  // Channel definitions connecting internal ports
  channel [controller.priorityControl] -o)-
    [priorityLight.control]
  ...
}
```

2) Asynchronous Components: Asynchronous components represent independently running instances that communicate with *messages*, collected in *message queues*. There is no guarantee on the execution time or the execution frequency of components. There are two types of asynchronous components in Gamma: *asynchronous composite components* and *synchronous component wrappers*.

a) Synchronous component wrapper: A synchronous component wrapper is used to declare an asynchronous component implemented by a single synchronous component, facilitating the hierarchical mixing of the composition variants. In addition to the ports of the wrapped component, synchronous component wrappers may declare additional ports for control messages, as well as zero or more *clocks*, which emit *tick* events at defined timed intervals.

A synchronous component wrapper has one or more *message queues*, which have the following attributes: *capacity* specifies the maximum number of messages that can be stored in the particular queue; *priority* specifies the order in which message queues are processed during the execution of the asynchronous component (the next message is always retrieved from a non-empty queue with the highest priority); *event types* specify the messages that will be put in the particular queue. Messages can be referred to either by specifically naming an event on a port, referring to all events on a port (with the *any* keyword instead of an event), or specifying the name of

a clock to refer to its tick event.

During execution, messages are retrieved individually from messages queues. A message is always taken from the highest priority non-empty queue. If the particular message was received on a port that is implicitly derived from the wrapped component, the message is converted to a signal (as synchronous components communicate with signals) and transmitted to the wrapped synchronous component (potentially overwriting previously sent signals). Otherwise, the message is not transmitted.

A synchronous component wrapper also has one or more *control specifications*, which describe when and how to execute the wrapped component. The trigger messages can be specified by referring to events as in case of message queues, while the execution mode can be one of the following: *"run once"* (*run* keyword) to trigger a single execution of the wrapped component; *"run to completion"* (*full step* keyword) to repeatedly execute the wrapped component until no more internal signals are generated (applicable only to composite components); *reset* (*reset* keyword) to reinitialize the wrapped component to its initial state.

The following code snippet presents an example wrapper for the previously defined *CrossroadComponent*, which defined a control port named *execution*. Messages received on this port are stored in the higher priority *executionQueue*, whereas the messages of additional (implicit) ports are stored in the other queue *crossroadsQueue* which has a capacity of 5. According to the control specifications, upon processing an *execute* message, the wrapped component is run to completion, while either a clock signal or an *interrupt* signal of port *police* triggers a single execution.

```
async AsyncCrossroad of CrossroadComponent [
  // Additional control ports
  port execution : provides Executable
] {
  // Clock definitions
  clock clockSignal(rate=100ms)
  // Control specifications
  when execution.execute / full step
  when clockSignal / run
  when police.interrupt / run
  // Message queues
  queue executionQueue(priority=2) {
    execution.execute, clockSignal
  }
  queue crossroadsQueue(priority=1, capacity=5) {
    police.any, priorityLight.any,
    secondaryLight.any
  }
}
```

b) Asynchronous composite component: Asynchronous composite components support the hierarchical definition of asynchronous components, just like synchronous composite components do for the synchronous case. The structure of asynchronous composite components is identical to their synchronous counterparts, but the type of constituent components must be asynchronous, i.e., either synchronous component wrappers of other composite components. The execution semantics of constituent components is totally asynchronous, any component may be executed any time, given that their message queues are not empty. Produced messages are placed into message queues of components on the other end of channels, and there is no restriction on how ports are connected. Note that the hierarchy of asynchronous components may always be flattened without affecting the meaning of the model.

IV. Conclusion and Future Work

In this paper, we have proposed an extension to the composition language of the Gamma Framework to support the mixing of three semantic variants for the composition of reactive components: asynchronous-reactive, synchronous-reactive and cascade semantics. Asynchronous components represent independently running components, which communicate with messages stored in message queues. This semantics is suitable for designing distributed or parallel processes. Synchronous-reactive components are useful for modeling a single executing unit consisting of multiple, functionally decomposed components. This composition mode is for the design of different aspects of a complex, but single-threaded component. Cascade composition is practical for designing units with pipeline-like behavior: the input fed into the model is processed by multiple consecutive filters in a single run.

Subject to future work, we plan to extend the code generation and formal verification services of Gamma to support the proposed language and the semantic variants it introduces. For code generation, we already have code templates for the more significant methods. The formal analysis of asynchronous-reactive models, however, will be worth more research, as the interleaving semantics of asynchronous models poses a serious challenge to model checkers that should be handled in the transformation to the formal input models of these tools.

Acknowledgment

Partially supported by the ÚNKP-17-2-I and ÚNKP-17-3-I New National Excellence Program of the Ministry of Human Capacities.

References

[1] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, Jun. 1987.

[2] B. Graics, "Model-Driven Design and Verification of Component-Based Reactive Systems," Bachelor's thesis, Budapest University of Technology and Economics, 2016. [Online]. Available: https://gamma.inf.mit.bme.hu

[3] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, Y. Xiong, and S. Neuendorffer, "Taming heterogeneity - the ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003. [Online]. Available: http://chess.eecs.berkeley.edu/pubs/488.html

[4] C. Brooks, "Ptolemy II: An open-source platform for experimenting with actor-oriented design," February 2016, poster presented at the "https://www.terraswarm.org/conferences/16/bears/" 2016 Berkeley EECS Annual Research Symposium. [Online]. Available: http://chess.eecs.berkeley.edu/pubs/1166.html

[5] S. Yovine, "BIP: Language and tools for component-based construction," 2007. [Online]. Available: http://users.cs.york.ac.uk/~burns/papers/bip.pdf

[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*. ACM, 2009, pp. 77–86.

[7] S. T. Karris, *Introduction to Stateflow with Applications*. Orchard Publications, 2007.