

Design and Specification of Concurrent System Components

Prakash Chandrasekaran
Chennai Mathematical Institute
prakash@cmi.ac.in

In a multi-component system with asynchronous communication, it is a challenge to specify the system formally, but in a style that is close to the techniques that engineers use. Ensuring that an implementation of such a system conforms to the specifications, is another challenge altogether. Widely used specification languages like UML are not rigorous enough to be used for formal verification. While formal models like Message Sequence Charts (MSCs) and High-level MSCs (HMSCs), are unable to represent the asynchronous nature of the communication. There have been some attempts at doing this in the form of Netcharts[6], Causal MSCs (CaMSCs)[5] and Compositional MSCs (CMSCs)[4]. But, these notations are not quite useable for practioners, and also not easy to interpret.

In practice, most asynchronous systems are implemented as event-driven systems. These are not only hard to reason about, but also hard to verify as a single logical action gets split across many different functions. Verification of this will involve pointer and heap analysis, which is beyond the scope of many of the existing verification tools.

In this paper, we'll briefly look at my work that addresses both the issues discussed above, and look at directions for future research in. We'll first look at CLARITY [2] – a language for writing asynchronous programs, and then discuss COORDINATED CONCURRENT SCENARIOS [3] – a formal model for local specification of communicating concurrent components.

Programming Asynchronous Systems

CLARITY is a new programming language, that extends 'C' with constructs that allow programmers to write asynchronous event based programs in a sequential manner. CLARITY introduces new language constructs “WaitFor” and “Async” that enable programmers to write asynchronous code sequentially. In addition, it also introduces the concept of high-level coordination constructs called “coords” and the notion of “linearity annotations”.

For example, in a device driver code, it is often the case that the device will take time to respond to the commands, and when this happens the driver saves it is state on the heap and returns control to the operating system. Later, when the driver is invoked again – say to service an interrupt – it then checks in it is queue and completes the pending request. Now, this means that a logical action – reading data from disk, say – is split across many functions in the driver code. This makes it difficult to write, manage, and verify the code. With the WaitFor construct, the programmer will just add an instruction to wait for the device to complete the task, and proceed with rest of the computation in the same sequential block.

In the following code snippet, the semantics of the WaitFor call ensures that the rest of the task is queued up and executed later when condition `Hardware->done` becomes true.

1 Complete presentation of CLARITY and CCS are available in [2] and [3].

```

/* Syntax is simplified to illustrate the basic idea alone. */
void* read(...){
/* do some pre-processing */
Hardware->read(....,value);
WaitFor(Hardware->done);
/* some post-processing */
return value;
}

```

The `Async` construct lets the programmer specify that a specific piece of code is to be executed in parallel (ie. asynchronously) with rest of the code. For example, polling for a device status is an activity that is by nature asynchronous to the main computation. In traditional programming models this polling is usually done at many points in the code, resulting in code duplication and fragmentation. In contrast, in `CLARITY` the programmer can write an `async` function that does the polling and can indicate the result to the primary thread of execution.

Designing Asynchronous Systems

In this section we will look at `ccs`, a formal model that helps specify the components locally, and enables modelling of concurrency and asynchrony in a natural manner.

A specification of a component p in `ccs`, is a Finite State Machine (FSM) whose edges are labelled by p -local MSCs. A p -local MSC represents only communication that involves the component p . We also allow the component to asynchronously execute MSCs, by annotating the transition with `async`. These MSCs are executed in parallel with rest of the system, and the FSM can proceed to the end state of the transition. An asynchronous MSC runs on its own thread, has no notion of a state, and terminates upon completion of the MSC.

The actual communication across the components are arrived at by composing the local MSCs, as guided by a given set of Transactions. A Transaction can be viewed as a “communication phase”, in which each component executes a collection of local MSCs, that combine to form a valid global scenario. For example, in a client-server setup, there can be a transaction T in a client server system that says that the Server executes the MSCs `ManageClient` and `ServeClient` concurrently, and the Client executes the MSCs `Login`, `Request`, and `Logout` in sequence.

T : Server(ManageClient || ServeClient) ; Client(Login;Request;Logout)

In this example above, we can also rewrite the transaction T as two transactions:

T1 : Server(ManageClient) ; Client(Login;Logout)

T2 : Server(ServeClient) ; Client(Request)

This example also illustrates another important feature of our model, namely the ability to specify a logical action whose parts are separated in time as a single MSC. This helps in static verification of many properties that might otherwise be infeasible to statically analyze. In our above example, one can easily verify that every resource allocated by the server on a client's login is freed upon it is logout.

In addition to this, `ccs` also defines a simple programming language over the *local* MSCs, that gives the model the power to efficiently model event-based systems. This language uses only three language constructs `IF`, `ATOMIC`, and `AWAIT` to enrich the *local* MSCs. The `AWAIT` is very much like the `WaitFor` in `CLARITY` and helps specify blocking conditions inside a MSC.

Impact on Verifiability

The ability to implement a single logical transaction as a sequential block, and to specify the same as a single scenario, means that no special pointer analysis or heap analysis is required to verify properties. This also enables static verification of CCS and CLARITY. While CLARITY can be easily verified by tools like SLAM and ZING, CCS specifications can be encoded into UPPAAL[1] templates, which can then be easily verified.

Putting The Pieces Together

One of the future directions, that I'm looking at, is bridging CLARITY and CCS to develop a complete framework, wherein the specification in the formal model can be used to generate CLARITY programs. The basic idea here is that the messages in the MSCs will correspond to API calls between the components, that are specified in a machine-readable form. These mappings, will be used to (partially) synthesize the implementation in CLARITY. Fully automated synthesis might still be a long way off. But, partial synthesis – like code templates generated by RAD tools – is something that appears to be quite achievable in the foreseeable future. Independently, the expressives of CCS could enable use of CCS or an adaptation to serve as a meta-language to model communication across different components of a heterogeneous system specified in various formal notations. Ongoing, prototype development in CLARITY suggests that it has the potential to become a useful language to develop device drivers, and other event-driven systems.

References

- [1] G. Behrmann, A. Davida, and K. Larsen. A tutorial on uppaal. *In SFM, volume 3185 of LNCS, pages 200–236, 2004.*
- [2] P. Chandrasekaran, C. Conway, J. Joy, and S. Rajamani. Programming asynchronous layers with clarity. *In FSE, pages 65–74, 2007.*
- [3] P. Chandrasekaran and M. Mukund. Specifying interacting components with coordinated concurrent scenarios. *Technical report, Chennai Mathematical Institute, 2009.*
- [4] B. Genest, D. Kuske, and A. Muscholl. A kleene theorem for a class of communicating automata with effective algorithms. *In DLT, volume 3340 of LNCS, pages 30–48, 2004.*
- [5] T. Gazagnaire and B. Genest and L. Helouet and P.S. Thiagarajan and S. Yang. Causal Message Sequence Charts. *In CONCUR, volume 4703 of LNCS, pages 166-180, 2007.*
- [6] M. Mukund, K. N. Kumar, and P. Thiagarajan. Netcharts : Bridging the gap between hmscs and executable specifications. *In CONCUR, volume 2761 of LNCS, pages 293–307, 2003.*