Analyzing Data Flows of State Machines

Julio Cano, Ralf Seepold, Natividad Martinez Madrid
Dto. Ingeniería Telemática, Universidad Carlos III de Madrid
Leganes, 28911, Madrid, Spain

ABSTRACT

One of the key points in Model-Driven Development is to provide a semantic anchoring that permits to design an application based on some common semantics but at the same time independently of the specific characteristics of the final platform.

This paper proposes a common meta-model capable of holding both state machine and data flow semantics, two of the most used behaviour models. This is done so that application behaviour can be described independently of the specific platform. The application behaviour can be based on the semantics of this meta-model while at the same time code can be generated for several specific platforms without changing the application design.

1. INTRODUCTION

Semantic anchoring is the base for model transformations [1], especially in a MDD (Model Driven Development) process, where automatic model transformations are highly desirable when possible to automatize the whole development process.

This work is centred in the behaviour modelling part of the application design. The most extended models in the literature for behaviour modelling are state machines and data flow diagrams. The statechart type of state machines as described in the UML Superstructure specification are used here due to its extended semantics compared to the semantics of a basic state machine.

In some cases [2] it is shown that only one behaviour meta-model is not enough to easily model the solution to a problem and it has to be modelled using one point of view but implemented in a different one. Alston and Madahar propose a solution to their problem easily modelling it using a state machine but this state machine is implemented using a data flow language.

This problem shows that a multi-view modelling capability to design applications is highly desirable, having the possibility to choose the most suited view to solve every problem. Some proposals are made to solve this problem, like in [3], where multiple computational models can be used to create heterogeneous compositions of mixed computational models. This solution permits to solve every part of the system using a different computational or behavioural model, but this model cannot be changed because the semantics of the different computational models are still separated.

What is needed is a multi-view solution where semantics are common to all the different views, allowing to change the view but maintaining the model semantics, as proposed in this work, using state machines and data flow diagrams.

State machines are generally used to describe the behavior of software components. They can represent the state of the component or application at every moment during the execution of a program or system, as well as the actions to be taken depending on the current system state and events received. No special emphasis is put on how data is treated through the application components, although state machines are considered to be a subset of Petri Nets.

On the other hand data flow diagrams are used to represent the behavior of a system too, but based on how data is moved from component to component and processed. Nothing is said about the state of the system at any point in the time of execution of the system, nor how should the system respond to received events, especially asynchronous ones.

However, both aspects of the behaviour of a system (how data is processed through the system as well as how the system responds to events depending on the current state) are essential in the system design.

In order to achieve that objective the main aspects of both computational models are analyzed here looking for the common points that could let a designer to see a system from both points of view at the same time. State machines and data flow diagrams will be analyzed next to find the common points.

Supposing that the system is to be designed using components, a multi-view approach is useful to model all the system in a hybrid point of view. To provide such multi-view modelling possibility permits to the different kind of developers to work on their point of view of the system and at the same time keeping a coherent model representing the system [5]. For instance, software components can be represented as data flow nodes given their interface nature with input and output channels, while at the same time internal behaviour of components can be represented as state machines or data flow diagrams.

The main objective of this work is to provide a common meta-model unifying both computational models making it possible to design a system taking into account both event responding and data flow processes in the system. In this paper a different approach to the one of Broy is used, less formal and more oriented to its implementation in MDD development tools.

The next section describes the software component model in which this work is based. Section 3 describes the states and data flow models that are analyzed. Section 4 compares both models analyzing the possibilities to convert from one model to another and the use of a common meta-model. Section 5 proposes a common meta-model based on the previous analysis and section 6 concludes this work with some description of future work.

2. SOFTWARE COMPONENTS (INTERFACE MODEL)

One of the most often used architectural (structural) design view is the Component Model. Being an essential part in the UML standard makes its use very extended too. The Component Model presented in the UML standard is related to the Interface Model, given that emphasis is made in the interfaces of the components. This Interface Model results close to the SOA (Service Oriented Architecture), being both based in the description of provided services to the rest of the system.

In [4] several different kinds of component models are described. One dimension on which components are categorized is depending on composition at design phase and other is at deployment phase. Components can be described independently at design phase, but instances of components in the application have to be connected to other instances at the deployment phase. It can be noted that independently of when component composition is done connection through interfaces results into a data flow diagram.

In summary, component composition can be described using the interfaces model, a dataflow model and even a state machine model.

In this paper it is supposed that a component model is used for the structural description of the application so that the behavioural description of the application is done inside the components.

3. MULTI-VIEW MODELLING

Two of the most common behaviour modelling views used in literature and software development in general are the states machines and the data flow diagrams. The subsection describe the most relevant models for the presented approach

A. States model

The state machines model is, probably, the most extended computational model due to its use in the UML standard. Basic state machines lack of some semantic elements needed for this work, so statecharts are used instead. The possibility to use *regions* to simplify the complexity of a state machine is definitely helpful in the conversion from data flow models to statechart models. In this work regions are considered to work in parallel where different transitions can be activated and executed simultaneously in separated regions.

Here it is a somewhat simplified state machine specification based on OMG UML 2.0 Superstructure Specification, including regions (pseudostates are eliminated for clarity purposes):

```
SM = {Regions}
Regions = {Transitions, States}
Transitions = {Trigger, Constraint, Behaviour,
S1, S2}
State = {Constraint, Trigger, entryBehaviour,
exitBehaviour, doActivityBehaviour}
```

Where `Trigger` represents the event that activates the transition, `Behaviour` is the action or process executed in the transition, `S1` and `S2` are the source and destination states of the transition respectively. In this case behaviours associated to the state are not taken into account (entry, exit and doActivity behaviour).

The use of regions is necessary (as described before) since state machines are compared to data flow diagrams which are parallel by nature.

B. Data Flow model

Data flow diagrams are, as described before similar to the components model, but with emphasis on the data flow and processing. The main characteristic used here is the behavioural or computational aspect instead of the architectural or structural one.

Here it is a simple data flow diagram specification:

```
DFD = {Processes, DataFlows, DataStore,
External}
```

Where `Processes` are considered to be equivalent to the `Behaviour` element in the state machines: it represents the action to be applied to the data. `DataFlows` are elements connecting `Processes`. It specifies the data goes from one process element to another. `DataStore` and `External` can be considered as specific `Processes` that store information in local or external areas.

In this paper data flow diagrams are considered to be parallels, where different process units in the same component can be activated independently. This will be taken into account in the possible transformations from data flow diagrams to statecharts.

4. MODELS ANALYSIS

These models do not seem to hold enough common base to have a common meta-model that supports both of them.

If system components are treated like black boxes a state can be assigned to a component depending on various elements. In a component designed with a data flow model the state can be represented depending on the state of the internal elements and data. The next can represent the component behaviour:

$$F: (s, I) \rightarrow (s', O)$$

Where `s` is the actual component state and `I` is the input to the component, while `O` is the output after the input has been processed and `s'` is the new state of the component.

Trying to enumerate and describe the number of possible states of a data flow diagram can require a great effort depending on the complexity of the model.

This work analyzes the way to translate every element of one model to another separately, so that step by step the transformation from one model to another can be affordable.

As seen in the previous section, there is a data processing element in both models (processing units and transition actions). The problem in state machines is that connection between these processing elements is not explicitly specified as in data flow diagrams. Data is transferred from one state to another as a trigger or event sent by a transition and received by a new active state.

Transitions can be represented as processing elements in a data flow diagram with input data being the event received by the state and the output data being the event generated by the transition.

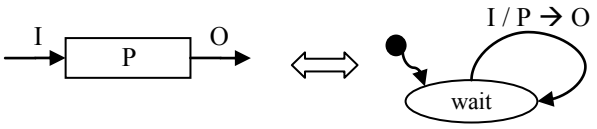

Fig 1 DataFlow and State Machine correspondence

On the other side, a data flow processing element can be analyzed from the state machine's point of view. Generally, every element can be considered to be waiting to receive input data to be processed. So this is a *wait* state that can be represented in a state machine. The input data is expected to be received with the incoming event. This event activates the transition that processes the input data. And after processing it the output data is sent out. Finally, the processing element returns to the wait state again. This means that the transition returns to the same wait state. This can be represented in a state machine as a transition from and to the wait state as in Fig 1.

The notation used in this work for the representation of the transitions in state machines is `trigger / action → event`. Where `trigger` is the event that activates the transition, `action` is the process or action to be activated by the transition and `event` is the output of the transition, generally a trigger that will activate another transition in the application.

A first step to convert a data flow diagram to a state machine could be to translate every processing element to a state machine containing one wait state and a transition to the same state. Every processing element could be contained in a separated region of a statechart so that independence and parallelism of every process can be represented.

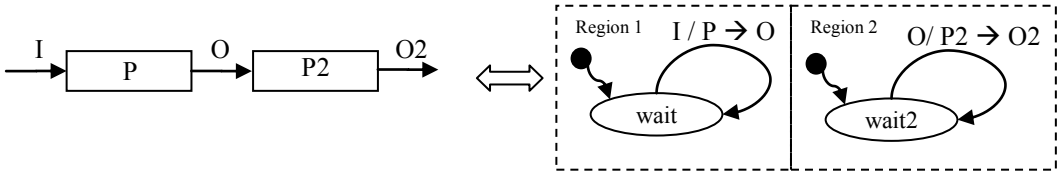

Fig 2 Multiple processing elements in a data flow diagram

Fig 2 shows how different processing elements in a dataflow diagram can be translated into different regions of the same statechart. In this case the output `O` of processing unit `P` is the input for the processing unit `P2`, but this fact is not strictly represented in the statechart. The state machine mechanism makes the output event `O` of the first transition to be received by the `wait2` state and be processed like in the data flow diagram.

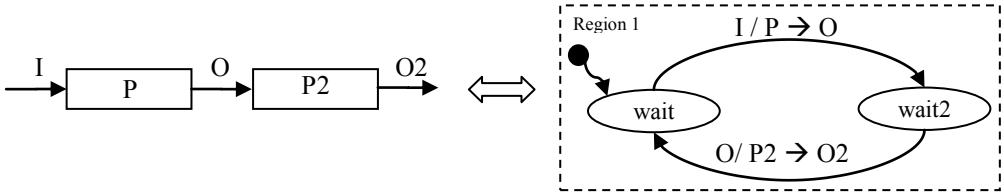

Fig 3 Data Flow between states

Taking into account that P is always connected to P2, the translation can be done so that the transition is made to the wait2 state, representing similar semantics to the data flow diagram ones. Being more specific, the conversion can be done using microsteps given that the data processing units are chained. As a result only one region is needed, simplifying the statechart (see Fig 3).

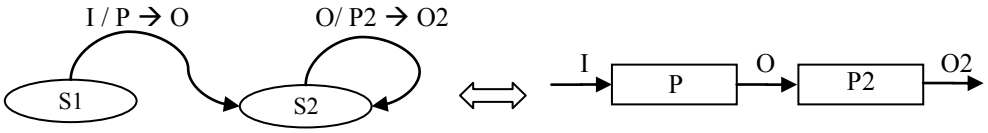

Fig 4 Chained data processing

As a result, state machines can be analyzed looking for data transferred between states. If data or an event is generated in a transition and received by other state and processed in one of its transitions then it can be translated to chained processing units in a data flow diagram as in Fig 4.

This kind of transformation is not so simple. If the state has several possible transitions then the second processing unit cannot be connected only to the first processing unit. The process activated will depend on the event. So a selection has to be made depending on the incoming event. Fig 5 shows an example of data flow diagram representing a state machine. Given that the state machine has a determined number of states and accepted input events a selection is made depending on them. Microsteps or chained transitions where the output of one transition is the input of the next transition can be transformed in chained processing elements. This kind of representation almost corresponds to the one in SDL (Specification and Description Language).

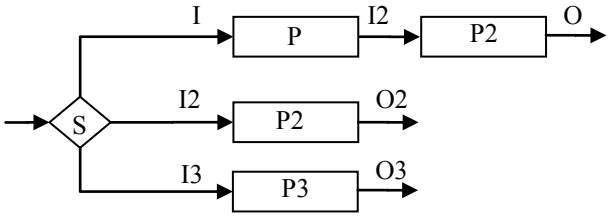

Fig 5 Example of data flow diagram representing a state machine

The use of this notation partially solves the problem of analyzing the state machine or statechart data flow. SDL notation has to be used, like indicating state changes in transitions, to keep information about actual state in the state machine model view. But still work has to be done to be able to fully represent a data flow diagram using a state machine model or a possible common meta-model that can represent both at the same time.

There is still a semantic problem to be solved with transitions and their activation events. One transition is activated when an event is received in a given state. Sometimes it is necessary to indicate that the same transition must be activated by different events or triggers independently. It can be represented by several different transitions having different activation events but with the same target state and effect. Common semantics and modelling tools accept a list of triggers or events that can activate a transition. But in some occasions it is required to indicate that more than one event is required at the same time to activate a transition. In a data flow diagram like the one in Fig 6 the processing unit P3 has different input values and all of them can be needed to obtain the corresponding output. This cannot be easily represented with one wait state and one transition at least that it can be specified that all of these incoming events are required for the activation. This would represent to store incoming events until all the required ones are available. In case that these semantics are not available a mechanism must be found to represent it using strictly statechart semantics.

A process unit is supposed here where all the incoming data elements are needed for the process to be executed but only one instance of each. Variations can be easily created depending on the specific semantics of the processing unit functioning. The first step is to be able to store incoming data. Supposing a data flow diagram with parallel nature, incoming data events will not arrive simultaneously. To be able to process and store every event independently a wait state is needed in an independent region for every incoming event. Given that only one instance is needed a second state is used to indicate that data has already been received. This way only the required events will be processed until all of these regions receive the corresponding signal to change again to the waiting state.

To know whether all required events have been received a counter can be used. This counter consists of a number of states indicating the number of events received. To activate the transition between these states the regions dedicated to receive and store incoming data will send the corresponding signal when an event is received. The final transition when all the needed data is available is responsible to execute the action corresponding to the process unit, return the counter to 0 and reset regions dedicated to receive events.

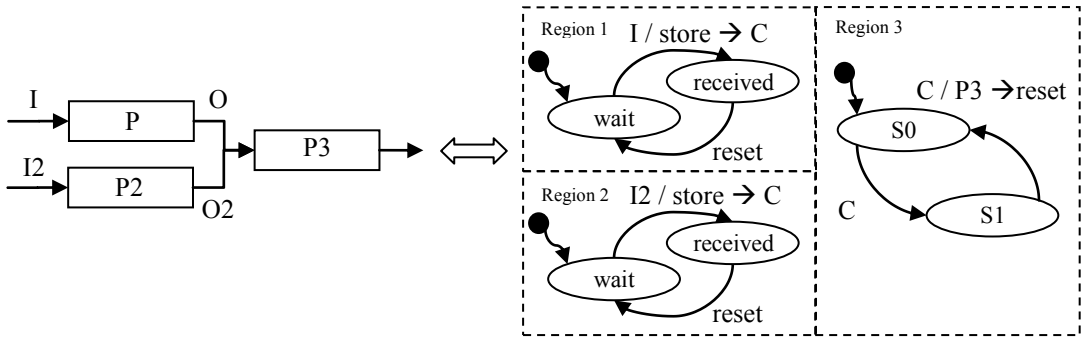

Fig 6 Multiple events for transition activation

5. COMMON META-MODEL

As shown in the previous section, it is possible to analyze data flow in state machines, generating equivalent data flow diagrams. On the other hand, data flow diagrams can be analyzed and statecharts can be generated representing the same semantics. To do that statechart regions have to be used to represent the parallel nature of data flow diagrams.

But to be able to represent both kinds of models at the same time and use them as complementary views a common meta-model is needed. A meta-model is proposed here based on the specifications used in section 3.

One of the basic issues is representing the data flow. To keep the states information and at the same time be able to connect data flows only information about these data flows is included in the state machine model.

```
SM = {Regions}
Regions = {Transitions, States}
Transitions = {S1, S2, Process (Input Events,
Guard condition, Output Events)}
```

The main changes regarding the previous model are that the Behaviour element is named Process after the data flow specification and it includes the guard condition for the transition activation (previously the constraint). The other changes are the addition of the specification of the list of input events or triggers that activate the transition and are supposed to be the input data for the process and the specification of the output data elements of the process element too. This information permits to connect the output elements of a process unit to the corresponding input of other process elements.

Here is a simple example applying the proposed meta-model to represent the models in Fig 2:

```
SM = {Region 1, Region 2}
Region 1 = {transition P}, {wait1}
Region 2 = {transition P2}, {wait2}
Transition P = {wait1, wait1, P (I, _, O)}
Transition P2 = {wait2, wait2, P2 (O, _, O2)}
```

Beside the states information, the data flow information can be extracted from the example. There are two process elements in the model (`P` and `P2`) both of them with their input and output elements. These elements can be connected because `O` is indicated to be the output of the process `P` and the input of the process `P2`. In this case no constraints are specified so a blank space is used in the model to indicate the absence of them.

This example shows that conversion from and to data flow diagrams and state machines can be done. A common meta-model based in this analysis allows representing state machines and data flows indistinctly. This meta-model permits an applications design tool to design the behaviour of application components as data flow diagrams or state machines indistinctly. Afterwards the point of view can be changed depending on the needs without changing the behavioural model of the application.

6. CONCLUSIONS AND FUTURE WORK

Semantic anchoring is an important part of an MDD process. Higher level model semantics must be based on lower level model semantics so that refinements between models can be implemented automatically.

In this work a simple common meta-model is proposed to help in the semantic anchoring of one of the weakest and complex areas in MDD: the behavioural modelling. This meta-model permits to represent simultaneously state machines (statecharts) and data flow diagrams. These behaviour meta-models are used here since they are the most widely used in the bibliography.

The proposed meta-model can hold at the same time the state machine and data flow model semantics providing semantic anchoring for upper levels of the behaviour design of applications, and at the same time allowing the designers to use different views of the application components depending on their specific needs.

Future work will concentrate on the specification of algorithms to work with this meta-model, to convert models from one view to another. At the same time an implementation of this meta-model will be developed to provide behaviour modelling at a Platform Independent level of a MDA process and code generation for a given specific platform.

7. ACKNOWLEDGMENTS

This research has been partially supported by the project OSAMI (Spanish Ministry of Industry, Tourism and Commerce: TSI020400-2008-114); Caring Cars (MEDEA+ 2A403) (Spanish Ministry of Industry, Tourism and Commerce: FIT-330215-2007-1) and InCare (Spanish Ministry of Education and Science: TSI2006-13390-C02-01).

8. REFERENCES

[1] Chen, K.; Sztipanovits, J.; Abdelwalhed, S. And Jackson E., *Semantic Anchoring with Model Transformations", ECMDA-FA 2005, LNCS 3748, pp. 115-129 2005.*

[2] Alston, I., Madahar, B., *'Controlling Data Flow Applications in Gedae: Is a Finite State Machine the Answer?'*, BAE SYSTEMS, 2003.

[3] Goderis, A.; Brooks, C.; Altintas, I.; Lee, E.A.; Goble, C.; *Heterogeneous Composition of Models of Computation*. Technical Report No. UCB/EECS-2007-139, Berkeley, November 27, 2007.

[4] Kung-Kiu L., Zheng W., *Software Component Models*, IEEE Transactions on Software Engineering, Vol. 33, No 10, October 2007.

[5] Broy M., *Multi-view Modeling of Software Systems*, LNCS 2757, pp. 207-225, 2003.