CSP Parallel Composition of Aspect Models

Ashley McNeile
Metamaxim Ltd, UK
ashley.mcneile@meta-maxim.com

Ella Roubtsova
Open University of the Netherlands
ella.roubtsova@ou.nl

ABSTRACT

We present an approach to aspect specification based on the Protocol Modelling paradigm, which uses CSP parallel composition as the mechanism for combining partial behaviours specifications. Using the Protocol Modelling approach enables local reasoning about the behaviour of the whole model based on knowledge of behaviour of the composed aspects, and we present the proof of this key property in this paper. We describe how Protocol Modelling allows the definition of aspect join points and weaving relationships at the model level, and how different aspects may use different abstractions over the same domain. We illustrate this using a small case example.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.1 [**Requirements. Specifications**]: []; D.3.1 [**Formal Definitions and Theory**]: []

Keywords

aspects, models, Protocol Modelling, CSP composition, join points, local reasoning

1. INTRODUCTION

Aspect technologies were first proposed as a programming technology by Kiczales et. al. [9]. Some recent work, though, has highlighted the value of considering aspects early in the life cycle as part of the basic strategy for modelling [3]. Thus far there is no standard way to present aspects at the modelling stage or standard mechanisms whereby such aspects may be woven together [14]. Many different modelling notations are used (state machines, Petri Nets, sequence charts, and process modelling languages such as BPML, PBEL) and each notation brings its own approach to representing aspects and to aspect weaving [8].

In line with this observation, this paper explores an approach to handling aspects at modelling stage, when the focus is on the identification of the appropriate behavioural modelling abstractions, before any consideration has been given to the physical design or production of code. We explore the possibility of providing a more abstract, and therefore more universal, definition of aspects and weaving by using the *notation independent* semantic framework offered by Protocol Modelling [1]. Protocol Modelling is an event based modelling paradigm in which complete behavioural models are built from components (called *Protocol Machines*) which are composed using the parallel composition operator of Hoare's process algebra, Communicating Sequential Processes (CSP) [4].

Its compositional approach makes Protocol Modelling a natural candidate for working with aspects. In this context we believe that Protocol Modelling has two advantages over other approaches:

- Its semantics is *notation independent* as it decouples the semantics of aspects and weaving from the choice concrete formalism used for modelling. This gives the modeller the freedom to select the most appropriate behavioural modelling formalism for a problem, and even to use different formalisms for different aspects within the same problem.

- It supports a useful form of *local reasoning*, whereby properties of the behaviour of the whole can be deduced from local knowledge of behaviour of the composed aspects. This is a requirement if the approach is to scale successfully to large and complex problems, as otherwise it is very hard for the modeller to retain intellectual control over the whole model as it grows.

In order to demonstrate Protocol Modelling, we use a small case study modelled as protocol machines rendered in state transition notation. Note, however, that the notational similarity to UML statecharts [19] is superficial: neither UML statecharts nor conventional statecharts [7] have semantics for event refusal [18], so *neither can support CSP composition* (for which the concept of event refusal is crucial). For clarity, we refer to this modified semantics as *PM-statecharts*. However, Protocol Models can be represented in any notation that can support the semantics of an event alphabet and the ability of a machine to accept or refuse a presented event. Previous work by the authors [20] demonstrates how the semantics of popular notations can be modified to support Protocol Modelling semantics.

This paper is structured as follows:

Section 2 defines the Protocol Modelling approach to modelling behavioural aspects and discusses local reasoning.

Section 3 discusses the specification of join points, in terms of event and state abstractions.

Section 5 compares our work with related research in the field.

Section 6 presents brief conclusions.

2. PROTOCOL MODELLING OF ASPECTS

2.1 Protocol Model of a Case Study

Our case study is a door with an alarm installation and event logging. A door can be opened, closed, locked and unlocked. The alarm system can be set and unset and, if set, goes into alarm if the door is unlocked or opened. When the alarm is set or in-alarm, an occurrence of any of {*Open*, *Close*, *Lock*, *Unlock*} must be logged in the alarm's memory.

We recognize three concerns: *Door*, *Alarm* and *Logging* and model each of them separately, as protocol machines rendered in PM-statecharts (Figure 1). The upper left diagram represents the *Door*. It uses three mutually exclusive states, {*Closed*, *Open*, *Locked*}, and represents the possible ordering of events that alter these states: for instance, you can only *Lock* the *Door* when it is *Closed*, and you cannot *Open* the *Door* when it is *Locked*. The upper right diagram, *Alarm*, represents an alarm device that is installed on the *Door*. This device goes into the state *InAlarm* (e.g., sounds an alarm bell) if an *Open* or *Unlock* event occurs while in the state *Set*. The lower diagram shows that the certain events are to be logged if the *Alarm* is *Set* or *InAlarm*.

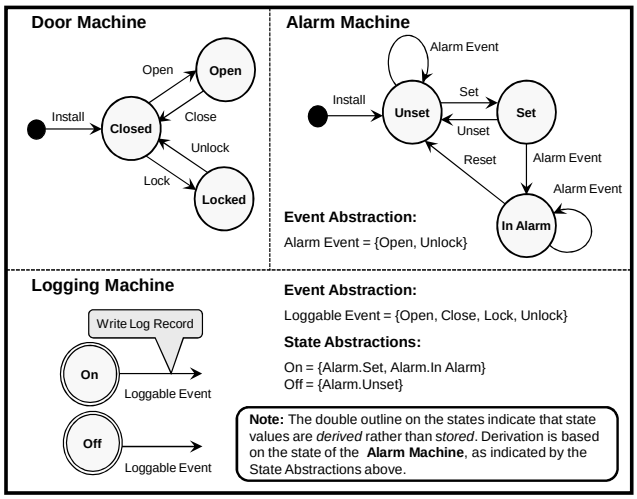

Figure 1: Door || Alarm || Logging in PM-statechart Notation

2.2 Protocol Machines

A *Protocol Machine* (hereafter referred by just "machine") is a conceptual event driven process. A machine has a defined alphabet of event types that it understands, and the ability to *accept*, *refuse* or *ignore* any event that is presented to it by its environment [1].

Events

The basis of defining the behaviour of a Protocol Model is identification of occurrences of interest in the environment (or domain). Examples of such occurrences in our case example are *"Install an Alarm on Door 345"* or *"Open Door number 456"*. These occurrences are taken to be atomic and instantaneous in the environment.

An "event" (more properly "event instance") is the data representation of an occurrence in the environment as a set of data attributes. Every event is an instance of an event type, and the type of an event determines its metadata (or attribute schema), this being the set of data attributes that completely define an instance of the event type. In our case study, an event might be:

```
Event Type = "Install",
Door Identifier = "456",
Alarm Identifier = "A1456".
```

Whenever an occurrence of interest takes place in the environment, a corresponding event is created and is presented to the machines of the model. How the events are created is not of concern in the modelling.

Between handling events, a machine rests in a well defined quiescent state, meaning that it can undergo no further change of state unless and until presented with a new event. A machine may only be presented with new event when in such a state. This approach to modelling occurrences in the domain as events is identical to that used in other event based modelling approaches [12, 15].

Alphabet

A machine has an *alphabet* of event types that it understands. For example, the alphabet of the *Door* machine is {*Intall*, *Open*, *Close*, *Lock*, *Unlock*}.

The alphabet of a machine is the set of labels on the statechart transitions.

Local Storage

A machine has *local storage* which only it can alter, and only when moving to a new state in response to an event. A machine may read, *but not alter*, the local storage of other machines with which it is composed. A machine has an initial state which its local storage takes when a machine is instantiated.

The local storage of a machine is represented as a set of attributes associated with the machine. For example, the *Logging* machine in Figure 1 maintains a log of significant events as part of its local storage. Rules shown in bubbles attached to transitions define how a machine updates the attributes of its local storage in response to an event, so the *Logging* machine adds a log record whenever that transition "fires".

When a machine is instantiated, all attributes of its local storage are given initial values according to their type, so the log record list structure in *Logger* would be created but empty. Machines that have a stored (as opposed to derived) state also have an implicit state marker in their local storage, and this is set to the black dot ("start state") when the machine is instantiated.

Behaviour

When a machine is presented with an event it will either *ignore* it, *accept* it or *refuse* it as follows:

- When a machine is presented with an event that *is not* in its alphabet, the machine ignores it.

- When presented with an event that *is* represented in its alphabet, it will either accept it or refuse it.
- Acceptance or refusal of an event by the machine is determined by rules that the machine evaluates based on the values of its accessible storage (i.e. its own local storage and the local storage of other machines composed with it) both *before* and *after* the event.

Note that "refusal" means that the machine is unable to handle the event at all, and this normally means that some kind of error message is generated back to the environment. How or where such an error is generated is not of concern for modelling purposes.

These behaviour rules are more general than, for instance, those represented by standard state-transition models such as UML statecharts. In particular, if behaviour is represented using PM-statecharts, the state can be *derived* on-the-fly rather than being stored as part of the machine's local storage. This is exactly analogous to the concept of a *derived attribute*, as defined in modelling approaches such as UML. For instance, an attribute *Age* might be derived from $DateOfBirth$ and $CurrentDate$ (the latter supplied by a clock) rather than being stored. The *Logging* machine in Figure 1 is an example of a machine that uses a derived state, derivation being based on the state of the *Alarm* machine. Note that machines with derived state do not need to be "topologically connected", as the new state that results from a transition firing is not determined as the end point of a transition; and also have no "start state" (black dot).

A *trace* of a machine is any ordered sequence events accepted by a machine starting from its initial state. For example, the sequence $\langle Install, Set \rangle$ is a trace of protocol machine *Door* $\|$ *Alarm* $\|$ *Logging*. Event $Install$ is accepted by both *Door* and *Alarm* because this event belongs to the alphabets of both machines and is accepted by both in their start states, but is not in the alphabet of *Logging* which therefore ignores it. *Set* is accepted by *Alarm* from the state $Unset$, but does not belong to alphabets of the other machines. The *behaviour* of a protocol machine is the set of all its traces.

The fact that machines have different alphabets, and a machine ignores events that do not belong to its alphabet, makes it possible to use separate machines to model separate behavioural aspects or concerns of a domain.

Determinism

Protocol machines are deterministic [1]. This means that:

- A machine is never presented with an event unless quiescent, and
- The new quiescent state that a machine reaches as a result of being presented with an event is completely determined by (1) the last quiescent value of its own local storage and the local storage of any other machines that it accesses, and (2) the event type and the attribute values of the event presented to it.

Machine behaviour is deterministic in the sense that executions of a protocol model are repeatable. If a given sequence of events is presented to a model twice, starting from the initial state, the final state will be the same. Determinism is not, however, guaranteed by the conventional semantics of statecharts. For this reason, rules need to be followed when modelling with a particular notation: with state-transition modelling, for instance, a machine must be constructed so that there is never more than one transition from a given state for a given event type.

Protocol Modelling, because it is based on parallel composition, is highly adept at modelling concurrency. The reason for requiring determinism is the ability this gives to argue about behaviour based on traces. There is, however, no assumption that non-determinism may not be introduced at physical design time if it is decided to distribute the model across multiple (real or virtual) processors; but this must be done in such a way that the behaviour of the model is not broken by such a distribution - discussion of this is beyond the scope of this paper.

2.3 Composition of Protocol Machines

Protocol machines have the property that large, complex machines can be assembled from small, simple ones. Composing two protocol machines yields a new protocol machine under the following definitions:

- The alphabet of the composed machine is the union of the alphabets of the constituent machines; and the local storage of the composed machine is the union of the local storages of the constituent machines.
- The rules for whether the composed machine accepts, refuses or ignores a presented event are:
  1. If both machines ignore the event, the composed machine ignores it;
  2. If either machine refuses the event, the composed machine refused it;
  3. Otherwise the composed machine accepts the event.

These rules correspond to the parallel composition operator (P‖Q) of Hoare's process algebra, Communicating Sequential Processes [4]. All the machines in a Protocol Model are composed in this way.

Objects

A population of objects is modelled using multiple instances of a given machine type, each having a unique *object identifier* to tie it to the object to which it belongs. For instance, to model a population of doors each with its own alarm the assembly *Door* $\|$ *Alarm* $\|$ *Logging* would be instantiated multiple times, each instantiation having its own identifier. If, on the other hand, we were modelling a configuration where multiple doors are managed by a single alarm the model would be: *Door*1 $\|$ *Door*2 $\|$ *Door*3 $\|$ *Alarm* $\|$ *Logging*. A fuller discussion of the approach to objects and their instantiation in Protocol Modelling can be found in [1].

2.4 Local Reasoning

One of the key concerns in aspect oriented modelling is that of "local reasoning" - namely the ability to reason reliably about the whole from examination of a part in isolation. If the use of aspect technologies results in specifications becoming distributed through multiple design artifacts (the base behaviour or code definitions and separate aspects that have been added to them) in such a way that no reliable deductions about the behaviour of the whole can be made from examination of a part, then there is little chance of maintaining intellectual control over complexity.

Local reasoning in Protocol Modelling is based on the following property of CSP composition: If we take a sequence, S of events that is accepted by the composition $(M_1 \parallel M_2)$ of the two machines M_1 and M_2, then the subsequence, S', of S obtained by removing all events in S that are not in the alphabet of M_1 would be accepted by the machine M_1 by itself. In other words, composing another machine with M_1 cannot "break its trace behaviour". This property is sometimes known as *Observational Consistency* [11] between a composite and its component machines.

We can use this property to support local reasoning thus: If the sequence S' were *not* acceptable to M_1, the original sequence S *could not* have been acceptable to $(M_1 \parallel M_2)$. This means that we can use properties of M_1 (or M_2) *alone* to argue about the behaviour of $M_1 \parallel M_2$.

The Observational Consistency property of CSP composition was established by Hoare [4]. However, Hoare's CSP formulation did not consider machines with derived states; so we provide a proof for composed protocol machines, covering the case of derived states, below.

Definition 1. A machine P is *trace independent* $\Leftrightarrow$ P does not read the local storage of another machine (so, in particular, does not use information from the local storage of other machines to determine to calculate a derived state). A machine that is not trace independent is *trace dependent.*

Definition 2. We denote the alphabet of a machine P by: $\alpha(P)$. A sequence, S, of events is a *trace* of a machine $P \Leftrightarrow$

1. All events in S belong to $\alpha(P)$ **and**:
2. **If** P **is trace independent:** If all events of S are presented in turn to P, they are accepted by P (i.e., no event in S is refused by P)

   or:

   If P **is trace dependent:** $\exists\ Q$, with $\alpha(Q) \subseteq \alpha(P)$, s.t. $(P \parallel Q)$ is trace independent and S is a trace of $(P \parallel Q)$.

Note that, by taking Q to be a machine that *ignores all events presented to it*, the second of part of the above definition (for the case P is trace dependent) is *also true* if P is trace independent.

Definition 3. We denote:

- the set of elements of $\alpha(P)$ involved in execution by P of a sequence S of events by: $\alpha(P) \upharpoonright_S$
- a sequence S of events restricted to the alphabet of a machine Q (i.e., the sequence obtained by removing from S every element that does not belong to the alphabet of Q) by: $S \upharpoonright_{\alpha(Q)}$
- the set of traces of a machine P by: $traces(P)$
- the set of traces of a machine P in which each element has been restricted to the alphabet of Q by: $traces(P) \upharpoonright_{\alpha(Q)}$.

Definition 4. A machine P is *Observationally Consistent* with a machine $Q \Leftrightarrow traces(P) \upharpoonright_{\alpha(Q)} \subseteq traces(Q)$.

Theorem. A Protocol Machine is Observationally Consistent with its components.

Proof. We wish to show that:

$$traces(M_1 \parallel M_2) \upharpoonright_{\alpha(M_1)} \subseteq traces(M_1)$$

Suppose that $T \in traces(M_1 \parallel M_2)$. There are two cases to consider:

M_1 is trace independent. In this case, we know that M_1 must either have accepted or ignored every event in T because, had it refused an event, that event would also have been refused by $(M_1 \parallel M_2)$ (by the rules of CSP composition) and T would not then be trace of $(M_1 \parallel M_2)$. As M_1 is trace independent, it is unaffected by the presence of M_2 in composition, and will accept/ignore events of T in the same way when executing alone. Moreover, the execution of M_1 alone is unaffected by events not in $\alpha(M_1)$ as it ignored these. So $T \upharpoonright_{\alpha(M_1)} \in traces(M_1)$.

M_1 is trace dependent. In this case, $M_1 \parallel M_2$ may be either trace dependent and trace independent. However, in either case, we know (by Definition 2) that $\exists\ Q$ s.t. $(M_1 \parallel M_2 \parallel Q)$ is trace independent and has T as a trace.

Now suppose that the local storage of $(M_2 \parallel Q)$, which we will denote by Σ, has value B_i before the i^{th} event of $T \upharpoonright_{\alpha(M_1)}$ and A_i after the i^{th} event of $T \upharpoonright_{\alpha(M_1)}$. Using these storage values, we construct a new machine, Q', to simulate to M_1 the storage environment provided by $(M_2 \parallel Q)$ through execution of the event sequence T, as follows:

1. $\alpha(Q') = \alpha(M_1) \upharpoonright_T$.
2. Q' is trace independent (i.e., we construct it to make no access outside its own local storage).
3. Q' accepts all events presented to it (i.e., we construct it to make no refusals).
4. Q' contains in its local storage a structure Σ' that mirrors Σ and provides an identical API to M_1 for access (so that M_1 cannot differentiate between accessing Σ and accessing Σ').
5. The value of the local storage Σ' is a derived (calculated on-the-fly) by Q'. The algorithm used by Q' for this derivation gives Σ' the value B_j if invoked to give the pre-state of the j^{th} event of $T \upharpoonright_{\alpha(M_1)}$, and the value A_j if invoked to give the post-state value of this event.

With this construction, Σ' simulates *exactly* the local storage values that M_1 would access from Σ before and after each event that M_1 processes (i.e., before and after processing each event in $T \upharpoonright_{\alpha(M_1)}$). Thus we have constructed Q' so that:

- $(M_1 \parallel Q')$ is trace independent. This is because $(M_1 \parallel M_2 \parallel Q)$ is trace independent so Q', by offering the same storage environment to M_1, must fully resolve every access that M_1 makes outside its own local storage. Moreover, we have constructed Q' to make no accesses outside of its own local storage.
- $\alpha(M_1 \parallel Q') = \alpha(M_1)$. This is because $\alpha(M_1 \parallel Q') = \alpha(M_1) \cup \alpha(Q')$, and $\alpha(Q') \subseteq \alpha(M_1)$ by construction.

- $(M_1 \parallel Q')$ accepts all the events of $T \upharpoonright_{\alpha(M_1)}$. This is because:
  - Q' provides M_1 with storage values (in Σ') that are identical to those it sees (in Σ) when consuming the event sequence T while composed with $(M_2 \parallel Q)$; and
  - Q' is constructed not to refuse any event presented to it.

These mean that $T \upharpoonright_{\alpha(M_1)}$ is a trace of $(M_1 \parallel Q')$ and so Q' satisfies the existence criterion required by Definition 2 to establish that $T \upharpoonright_{\alpha(M_1)} \in traces(M_1)$.□

3. JOIN POINT SPECIFICATION

IN this section we discuss how the Protocol Modelling approach allows us to specify the abstractions needed for join point specification and quantification.

If an event is in the alphabet of two composed machines, and they both accept it, they both engage in the handling of the event. So if an event type belongs to the alphabets of two machines, this event type is already a *join point* of the two machines. However aspects often address different levels of abstractions within a problem, and very often distinctions that are relevant to one aspect are not relevant in another. Supporting different abstractions in different aspects requires quantification in one aspect over the abstractions defined in another aspect. In this section we discuss the mechanisms that Protocol Modelling provides for defining different abstractions in different aspect machines and using these for join point quantification.

Event Abstractions as Join Points

The first type of abstraction supported by Protocol Modelling is by *event*. If all the events in a subset of the vocabulary of a machine are treated identically in the context of that machine, then it is possible to abstract from these multiple events by replacing them by a single alias in the definition of the machine.

For example, the events *Open* and *Unlock* of the *Door* machine are treated identically in *Alarm* machine and have been replaced by the alias *AlarmEvent*. Similarly, the events *Open*, *Close*, *Lock* and *Unlock* are treated identically in the *Logging* machine so have been replaced by the alias *LoggableEvent*.

Note that in the PM-statecharts notation, an arrow between two states labelled with an event alias is semantically identical to a set of arrows between the same two states, each labelled with one event from the set that the alias names.

State Abstractions as Join Points

The second type of abstraction supported by Protocol Modelling is by *state*. This exploits the fact that a machine is able to access (read only) the local storage of composed machines and can, if desired, use this to derive (or calculate) its state on-the-fly. This capability allows one machine to have a state that abstracts from the state of another.

The *Logging* machine is required to *log events* when the *Alarm* is either *Set* or *InAlarm*, but does not need to distinguish between these two states. It therefore uses more abstract states *On* and Off which are derived from the underlying states of the *Alarm* machine.

Combining Abstractions

The *Logging* machine uses both event and state abstractions:

- abstract events, using the alias *LoggableEvent* for $\{Open, Close, Lock, Unlock\}$
- abstract states, using the derived states *On* for $\{Alarm.Set, Alarm.InAlarm\}$ and Off for $\{Alarm.Unset\}$.

The single arrow which has the $WriteLogRecord$ attached to it therefore "joins to" any of the following state-transitions in the *Alarm* machine:

$$\{Alarm.Set, Alarm.InAlarm\} \times \{Open, Close, Lock, Unlock\}.$$

Quantification, in general, can be specified as a set of pairs:

$$(state, event)$$

where each of *state* and *event* may be aliases. The join

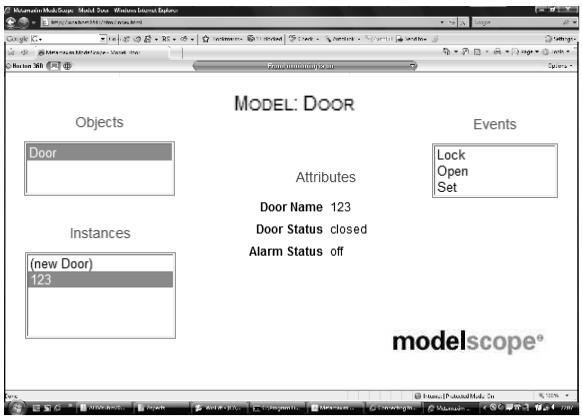

Figure 2: Modelscope User Interface

points are identifications between different machines that determine how the CSP composition is performed. It is this composition that we refer to as "weaving". The processing performed by different machines (i.e., the way different machines update their own storage to reflect the effect of an event) is independent between machines, not woven, so there is no idea of *code level weaving* (such as *before*, *after* and *around*) as found in code based aspect approaches [9]. In Protocol Modelling, it is the *behaviour of machines* that is woven, not the *representation of machines*.

4. EXECUTABLE PROTOCOL MODELS OF ASPECTS

The tool ModelScope [2] supports direct execution of Protocol Models, providing run-time aspect machine composition and a generic (metadata driven) user interface that allows events to be created and submitted to a model to test its behaviour. Figure 2 shows the ModelScope user interface as it appears executing the case example used in this paper.

5. RELATED WORK

Various elements presented in this paper relate to other work in the area of behaviour modelling in general and aspect modelling in particular.

There has been considerable work on the concept of Local Reasoning, for instance by Clifton and Leavens [5]; O'Hearn, Reynolds and Yang [13]; and Krishnamurthi, Fisher and Greenberg [16]. Pre- and post-conditions, frame axioms and invariants form the style of local reasoning used in the work of Clifton and Leavens. They have found that AspectJ in general does not support modular reasoning: the behaviour of a module can only be determined by an analysis of whole program. They propose a composition mechanism called explicit acceptance in which the base behaviour can accept an aspect under specified conditions. A particularly cogent argument for local reasoning, and the adoption of the term *Semantic Obliviousness* to describe the property of one part being oblivious of the effects of another is given by Dantas and Walker [6].

Kiczales and Mezini [10] introduce the concept of aspect-aware interfaces. They argue that "With aspect-aware interfaces we require a global analysis of the deployment configuration to determine module interfaces. But once that is done, modular reasoning is possible".

Katz [21] has described the categories of *spectative*, *regulative* and *invasive* aspects expressed in Aspect-oriented programming languages. The spectative aspects do not change the values of variables and the flow of method calls in the underlying system. Regulative aspects can affect the flow of control, and invasive aspects do change the values of variables of the underlying system. Protocol Modelling does not support invasive aspects as one machine cannot change the local storage of another, but spectative and regulative aspects can both be represented.

However all of the above work is based on the use of aspects in the context of programming, concentrating on the idea of composition as the weaving of executions of two *programs*, so do not achieve a lifting of the level of abstraction to behaviour *models*.

CSP composition is emerging as a valued technique in other areas. Grieskamp et al. working at Microsoft Research are proposing a use of it in the context of symbolic execution for program verification [17]. Their concept of *Action Machines* that can be composed using CSP bears an intriguing similarity to the concept of Protocol Machines.

6. CONCLUSION

In this paper we have investigated the application of CSP based process composition to aspectual style modelling, using the ideas of the Protocol Modelling approach. Our conclusion is that CSP parallel composition, combined with the semantics of abstract events and states, is capable of supporting useful join point specification; and allowing "local reasoning" about the whole behaviour based on knowledge only of a part.

7. REFERENCES

[1] A. McNeile, N. Simons. Protocol Modelling. A modelling approach that supports reusable behavioural abstractions. *Software and System Modeling* , 5(1):91–107, 2006.

[2] A. McNeile, N. Simons. http://www.metamaxim.com/.

[3] B. Tekinerdogan, M. Akşit. Deriving design aspects from conceptual models. *LNCS 1546*, pp. 587–588, 1998.

[4] C. Hoare. Communicating Sequential Processes. 1985.

[5] C. Clifton, G. Leavens. Spectators and assistants: Enabling modular aspect-oriented reasoning. *Technical Report 02-10, Iowa State University, citeseer.ist.psu.edu/clifton02spectators.html*, 2002.

[6] D. Dantas, D. Walker. Harmless advice. In *POPL*, pp. 383–396, 2006.

[7] D. Harel, M. Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach.* McGraw-Hill, 1998.

[8] D. Stein, S. Hanenberg, R. Unland. Expressing different conceptual models of join point selections in aspect-oriented design. *Proc. of AOSD 2006*, pp. 15–26, 2006.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin. Aspect-Oriented Programming. *Proc. of the European Conference on Object-Oriented Programming*, 1241: pp. 220–242, 1997.

[10] G. Kiczales, M. Mezini. Aspect-Oriented Programming and Modular Reasoning. *Proc. of the International Conference on Software Engineering*, pp. 49–58, 2005.

[11] J. Ebert, G. Engels. Observable or invocable behaviour-You have to choose. *Technical report. Universitat Koblenz, Koblenz, Germany*, 1994.

[12] M. Jackson, P. Zave. Domain Descriptions. *Proc. of the IEEE International Symposium on Requirements Engineering*, pp. 56–64, 1993.

[13] P. O'Hearn, J. Reynolds, H. Yang. Local Reasoning about Programs that Alter Data Structures. *LNCS 2142*, pp. 1–19, 2000.

[14] R. Filman, T. Elrad, S. Clarke, M. Akşit. *Aspect-Oriented Software Development.* Addison-Wesley, 2004.

[15] S. Cook, J. Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy.* Prentice Hall, 1994.

[16] S. Krishnamurthi, K. Fisher, M. Greenberg . Verifying Aspect Advice Modularity. *Proc. of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pp. 137–146, 2004.

[17] W. Grieskamp, F. Kicillof, N. Tillmann. Action Machines: A Framework for Encoding and Composing Partial Behaviours. *Microsoft Technical Report MSR-TR-2006-11*, 2006.

[18] S. Smith, D. Duke (1999). Using CSP to specify Interaction in Virtual Environments. *Technical Report YCS 321. University of York.*

[19] Object Management Group (2003). UML2.0 Superstructure: Final Adopted Specification.

[20] A. McNeile, E. Roubtsova (2007). Protocol Modelling Semantics for Embedded Systems. *Proceedings of the IEEE Second International Symposium on Industrial Embedded Systems, SIES'2007*, pp. 258–265.

[21] S. Katz (2006). Aspect Categories and Classes of Temporal Properties. *Transactions on Aspect-Oriented Software Development. LNCS 3880, Springer*, pp. 106-134.