A Formal Model for Cross-cutting Modular Transition Systems

Henny B. Sipma*
Computer Science Department
Stanford University
Stanford, CA. 94305-9045
sipma@cs.stanford.edu

ABSTRACT

We define a notion of aspects in the framework of modular transition systems. In our model an aspect is viewed as a semantic transformation on transition systems. Our primary objective is to use the model as a basis for studying inheritance and imposition properties of aspect constructs currently in use in practical languages such as AspectJ. We show that our model is sufficiently expressive to represent many of the constructs in this language. However, the mechanism of aspect-orientation presented in this paper may also be of practical use for systems organized in a modular fashion.

1. INTRODUCTION

In recent years cross-cutting techniques have emerged as a useful programming technique orthogonal to object-oriented and modular programming methods [7, 6]. In this paper we propose a formal model of such cross-cutting techniques, also called aspects, in the framework of modular transition systems, an expressive, first-order representation of reactive systems. Our aim is to use the model as a basis to study properties and capabilities of such techniques, including inheritance properties: which system properties are preserved across the application of aspects, and imposition properties: which systems are guaranteed to satisfy the property imposed by the aspect. Although not set in an object-oriented framework we expect our analysis results to give insight into the general application of aspect-oriented techniques.

In our model aspects are viewed as semantic transformations on modular systems. Aspects can introduce global and local state into modules, introduce additional statements, and modify existing statements, including altering the program flow. The system modifications are modeled by a combination of abstraction to add newly desired program behaviors, and restriction to remove unwanted program behaviors. Analysis of the effect of aspect application can thus make use of the well-known analysis methods for abstraction and restriction.

Although our main objective is to use the model as a basis for analysis, the application of aspects to modular transition systems presented here can also be useful in practical construction of reactive programs. Because of the semantic basis of our method, potentially more constructs are enabled than with current methods in which new code can be introduced only at so-called join points in the call graph of the program.

The paper is organized as follows. In the next section we introduce the computational model of transition systems and present a simple programming language to describe systems. In section 3 we define the representation and semantics of an aspect, and in section 4 we illustrate some typical aspect constructs commonly provided by aspect-oriented languages. In section 5 a preliminary outline is given of the types of analysis we expect to do based on the model presented here, and section 6 concludes with a discussion of some shortcomings of our model and plans for future work.

2. PRELIMINARIES

2.1 Computational Model: Transition Systems

Our basic computational model is that of a *transition system* [9] (TS), $\mathcal{S} = \langle V, \Theta_{\mathcal{S}}, \mathcal{T} \rangle$, where $V \subset \mathcal{V}$ is a finite set of typed variables taken from a universal set of variables $\mathcal{V}$, $\Theta_{\mathcal{S}}$ is an assertion (first-order formula) characterizing the initial states, and $\mathcal{T}$ is a finite set of transitions. A *state* s is an interpretation of $\mathcal{V}$, which assigns to each variable $v \in \mathcal{V}$ a value $s[v]$ over its domain; Σ denotes the set of all states. A transition $\tau \in \mathcal{T}$ is a function $\tau : \Sigma \mapsto 2^{\Sigma}$, and each state in $\tau(s)$ is called a τ-successor of s. We say that a transition τ is *enabled* on s if $\tau(s) \neq \emptyset$, otherwise τ is *disabled* on s.[1]

*This research was supported in part by NSF grants CCR-99-00984-001 and CCR-0121403, by ARO grant DAAD19-01-1-0723, and by ARPA/AF contracts F33615-00-C-1693 and F33615-99-C-3014.

[1]Note that the set of transitions can equally well be represented by a single transition. In the original definition of [9] separate transitions are identified to support the definition of fairness. Although we do not handle fairness in this paper, it is the intention to include it in the future. In addition, identification of separate transitions allows finer-grain control of aspect applicability.

Each transition $\tau \in \mathcal{T}$ can be described by a first-order formula $\rho_\tau(V, V')$, called the *transition relation*, expressing the relation between a state s and any of its τ-successors $s' \in \tau(s)$. In $\rho_\tau(V, V')$ the unprimed versions of the variables refer to values in s and the primed versions refer to values in s'. For example, the formula $x' = x + 1$ represents the transition function in which the set of τ-successors of a state s with $s[x] = c$ contains all states s' such that $s'[x] = c + 1$ for some constant c.

A *run* $\sigma : s_0, s_1, \ldots$ of a transition system $\mathcal{S}$ is an infinite sequence of states such that the following two conditions hold

- **Initiation:** the first state is initial, that is $s_0 \models \Theta_\mathcal{S}$;
- **Consecution:** for each $i \geq 0$, the state s_{i+1} is a τ-successor of s_i for some $\tau \in \mathcal{T}$.

The behavior of a system $\mathcal{S}$ is identified with its set of runs, denoted by $\mathcal{L}(\mathcal{S})$.

2.2 Modular transition systems

Modular transition systems organize a transition system into transition modules that can be composed into larger modules by means of module expressions [3]. In this paper we do not use the full power of module expressions, but restrict ourselves to modular transition systems consisting of n basic modules composed in parallel, where a basic module consists of an *interface* $I : \langle V_{input}, V_{output}, V_{shared} \rangle$ declaring the input variables V_{input}, which can be modified by other modules, but not by the module itself, the output variables V_{output}, whose value can be observed by the other modules, but can only be modified by the module itself, and the shared variables V_{shared}, which can be observed and modified by all modules, and a *body* $B : \langle V_M, \Theta_M, \mathcal{T}_M \rangle$ containing the set of variables local to the module, an initial condition, and a set of transitions. For the precise semantics of the parallel composition operator we refer to [3], as it is not directly relevant to the remainder of this paper.

2.3 SPL programs

Although systems can be described directly as modular transition systems, using first-order formulas to describe the initial condition and transition relations, it is usually more convenient to represent a system as a program in some structured programming language with a well-defined semantics in terms of transition systems. We will use SPL (Simple Programming Language) a simple imperative programming language to describe modular transition systems [9].

Example Figure 1 shows program BAKERY, a program that implements Lamport's *Bakery algorithm* for mutual exclusion [8], consisting of two SPL modules, P_1 and P_2. This program can be translated into a modular transition system with modules M_1 and M_2. The two modules have interfaces

$$\begin{array}{rcl} I_1 & = & \langle \{y_2\}, \{y_1\}, \emptyset \rangle \\ I_2 & = & \langle \{y_1\}, \{y_2\}, \emptyset \rangle \end{array}$$

respectively, reflecting that y_2 is observed by P_1, but not modified by P_1, and y_1 is modified by P_1, but not modified by the other module, and the other way around for module P_2. The bodies of the two modules are given by

$$\begin{array}{rcl} B_1 & = & \langle \{\pi_1\}, y_1 = 0 \wedge \pi_1 = \ell_0, \mathcal{T}_1 \rangle \\ B_2 & = & \langle \{\pi_2\}, y_2 = 0 \wedge \pi_2 = m_0, \mathcal{T}_2 \rangle \end{array}$$

where $\mathcal{T}_1$ and $\mathcal{T}_2$ contain the transitions corresponding to the statements in each module, as explained below.

Each statement in an SPL module is associated with a *prelocation*, identified by the label of the statement, and a set of *post locations*. To each module M_i a control variable π_i is added, ranging over the set of locations in the module, and initialized to the prelocation of the first statement in the module. Each statement corresponds with a transition according to the transition semantics of each statement. For example, the statement labeled by ℓ_2 is represented by a transition with transition relation

$$\pi_1 = \ell_2 \wedge (y_2 = 0 \vee y_1 \leq y_2) \wedge \pi_1' = \ell_3 \wedge y_1' = y_1$$

while the statement labeled by ℓ_0 can be represented by

$$\left(\begin{array}{c} \pi_1 = \ell_0 \\ \wedge \\ ((\mathit{true} \wedge \pi_1' = \ell_1) \vee (\mathit{false} \wedge \pi_1' = \ell_6)) \\ \wedge \\ (y_1' = y_1 \wedge y_2' = y_2) \end{array} \right)$$

In the remainder of the paper we will usually omit the conjuncts like $y_1' = y_1$, stating that a variable is preserved, and assume that all system variables not mentioned in the transition relation are preserved.

Thus, $\mathcal{T}_1$ and $\mathcal{T}_2$ each contain six transitions, one for each statement.

For a detailed description of the semantics of SPLin terms of transition systems, the reader is referred to [9].

$$P_1 : \left[\begin{array}{l} \textbf{out } y_1 : \textbf{integer where } y_1 = 0 \\ \textbf{in } y_2 : \textbf{integer} \\ \\ \ell_0 : \textbf{ while } \mathrm{T} \textbf{ do} \\ \quad \left[\begin{array}{l} \ell_1 : \textbf{ noncritical} \\ \ell_2 : \ y_1 := y_2 + 1 \\ \ell_3 : \textbf{ await } (y_2 = 0 \vee y_1 \leq y_2) \\ \ell_4 : \textbf{ critical} \\ \ell_5 : \ y_1 := 0 \end{array} \right] \\ \ell_6 : \end{array} \right]$$

$$\Big\| \quad P_2 : \left[\begin{array}{l} \textbf{out } y_2 : \textbf{integer where } y_2 = 0 \\ \textbf{in } y_1 : \textbf{integer} \\ \\ m_0 : \textbf{ while } \mathrm{T} \textbf{ do} \\ \quad \left[\begin{array}{l} m_1 : \textbf{ noncritical} \\ m_2 : \ y_2 := y_1 + 1 \\ m_3 : \textbf{ await } (y_1 = 0 \vee y_2 < y_1) \\ m_4 : \textbf{ critical} \\ m_5 : \ y_2 := 0 \end{array} \right] \\ m_6 : \end{array} \right]$$

Figure 1: Program BAKERY

In the examples in Section 4 we will use the functions *preloc* and *postloc* that assign to each statement its prelocation and a set of post locations, respectively, where the set of post locations is determined by the control flow of the statement, as defined by the transition semantics. For example, $preloc(\ell_1 : y_1 := y_2 + 1) = \ell_1$, and $postloc(\ell_0 : y_1 := 0) = \{\ell_1, \ell_6\}$, even though location ℓ_6 is not reachable via statement ℓ_0. Again these functions are fully defined in [9].

3. ASPECTS

An aspect is defined as a transformation that maps modular transition systems into modular transition systems. The modifications an aspect may make to a modular transition system include the addition of global and local state to the system and modules, respectively, the introduction of new transitions, and the modification of existing transitions[2]. An aspect is described by a set of global variables, possibly with an initial condition, and a list of aspect facets, one for each module constituting the modular system, specifying the additional state and modifications to each module.

Modification of transition relations is achieved by a combination of conditional abstraction and restriction. Transition relations that imply an abstraction condition are abstracted by projecting out a set of variables associated with the condition. Abstraction is followed by restriction: the transition relations of those transitions whose original transition relation implies a restriction condition are conjoined with the associated restriction assertion.

The approach of transformation is illustrated in Figure 2. The original set of behaviors of the system is enlarged by means of abstraction. This set is then reduced by restriction, possibly eliminating some or all of the original system behaviors.

Our definition of an aspect was inspired by the class extensions presented in [4], which are code facets that are added to a base system based on predefined entry and exit conditions in the base system.

3.1 Definition

An *aspect* $\mathcal{A} : \langle V_A, \Theta_A, \alpha_1, \ldots, \alpha_n \rangle$ is a transformation defined on modular transition systems $\mathcal{M} : \langle M_1, \ldots, M_n \rangle$ consisting of n modules, with

$$M_i : \langle \langle V_{i,input}, V_{i,output}, V_{i,shared} \rangle, \langle V_i, \Theta_i, \mathcal{T}_i \rangle \rangle$$

The aspect consists of the following components:

- V_A: a finite set of global aspect variables, disjoint from the variables in $\mathcal{M}$;
- Θ: the aspect initial condition, an assertion over V_A and the shared variables in $\mathcal{M}$;
- $\alpha_1, \ldots, \alpha_n$: a list of aspect facets.

[2]At this stage we do not allow aspects to modify the modular structure; it is a straightforward extension to allow aspects to introduce new modules.

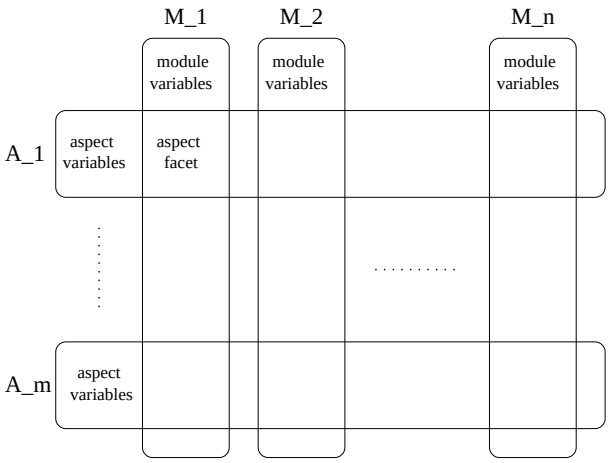


Figure 3: Schematic representation of aspect application

An aspect facet is defined for each module. It describes how the behavior of the module is modified. Figure 3 gives a schematic representation of the relationship between modules and aspects and the position of aspect facets.

An aspect facet $\alpha_i : \langle V_{\alpha i}, \Theta_{\alpha i}, \mathcal{T}_{\alpha i}, \beta_i, \epsilon_i \rangle$ consists of the following components:

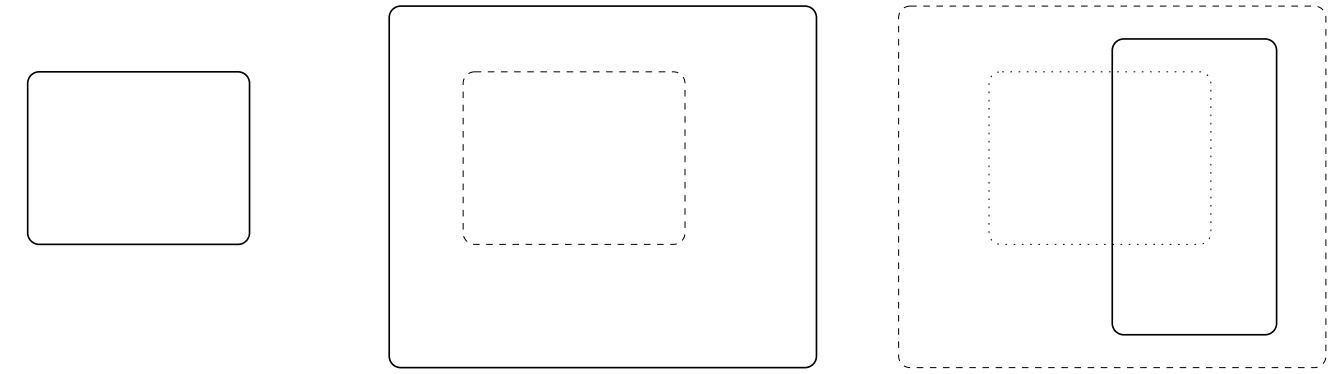
- $V_{\alpha i}$: a set of variables local to the facet, disjoint from the variables in $\mathcal{M}$; these variables are used to perform local bookkeeping and are not visible by other aspect facets.
- $\Theta_{\alpha i}$: the aspect facet initial condition, an assertion over $V_{\alpha i} \cup V_i \cup V_A$, used to initialize the local facet variables.
- $\mathcal{T}_{\alpha i}$: a finite set of transitions that may modify variables in $V_A \cup V_{\alpha i} \cup V_{i,output} \cup V_{i,shared} \cup V_i$, and in addition, may depend on the values of variables in $V_{i,input}$.
- β_i: a finite set of aspect facet abstraction instructions, where each instruction $\beta_{ij} = (\varphi_{ij}, W_{ij})$ is a pair consisting of an assertion φ_{ij} that governs the applicability of the instruction, and a set of variables W_{ij} to be projected out of the transition in case the transition relation implies the applicability condition;
- ϵ_i: a finite set of aspect facet restriction instructions, where each instruction $\epsilon_{ij} = (\psi_{ij}, \chi_{ij})$ is a pair consisting of an assertion ψ_{ij} governing the applicability of the instruction, and an assertion χ_{ij} to be conjoined with the transition relation in case the transition relation implies the applicability condition.

3.2 Semantics

Given an aspect $\mathcal{A} : \langle V_A, \Theta_A, \alpha_1, \ldots, \alpha_n \rangle$ and a modular transition system $\mathcal{M} : \langle M_1, \ldots, M_n \rangle$ the application of $\mathcal{A}$ to $\mathcal{M}$, written $\mathcal{A}(\mathcal{M})$, defines the modular transition system $\mathcal{M}^* : \langle M_1^*, \ldots, M_n^* \rangle$ with

$$M_i^* : \langle \langle V_{i,input}^*, V_{i,output}^*, V_{i,shared}^* \rangle, \langle V_i^*, \Theta_i^*, \mathcal{T}_i^* \rangle \rangle$$

with the following values



a) original set of behaviors b) set of behaviors after abstraction c) set of behaviors after restriction

Figure 2: An aspect as a combination of abstraction and restriction

- The input and output variables of all modules remain unchanged:

$$V^*_{i,input} = V_{i,input} \quad \text{and} \quad V^*_{i,output} = V_{i,output}$$

- The global aspect variables are included in the shared variables of each module, making these variables visible to all modules:

$$V^*_{i,shared} = V_{i,shared} \cup V_A$$

- The local variables of each aspect facet are included in the local variables of the corresponding module:

$$V^*_i = V_i \cup V_{\alpha i}$$

- The global aspect initial condition and the aspect facet initial condition are both conjoined with the module initial condition:

$$\Theta^*_i = \Theta_i \wedge \Theta_A \wedge \Theta_{\alpha i}$$

- The new set of transitions $\mathcal{T}^*_i$ consists of the module transitions, possibly modified by the facet abstraction and restriction instructions, and the facet transitions.

  Let $\tau \in \mathcal{T}_i$ be a module transition with transition relation ρ_τ, and let $\beta = \{(\phi_1, W_1), \ldots (\phi_k, W_k)\}$ be the set of abstraction instructions of aspect facet α_i. Then the abstracted transition τ_β has transition relation

$$\exists X \;.\; \rho_\tau$$

  where

$$X = \bigcup_{j \in I} W_j$$

  where I is the index set containing the indices of the abstraction instructions whose condition is implied by the transition relation, that is,

$$I = \{j \mid 1 \leq j \leq k \text{ and } \rho_\tau \rightarrow \phi_j\}$$

  Thus the variables associated with those abstraction instructions whose condition is implied by the transition relation are projected out of the transition relation.

  Similarly, let $\epsilon = \{(\psi_1, \chi_1), \ldots, (\psi_k, \chi_k)\}$ be the set of restriction instructions for aspect facet α_i. Then the abstracted and restricted transition $\tau_{\beta\epsilon}$ has transition relation

$$\exists X \;.\; \rho_\tau \wedge \bigwedge_{j \in I} \chi_j$$

  where

$$I = \{j \mid 1 \leq j \leq k \text{ and } \rho_\tau \rightarrow \psi_j\}$$

  is the index set containing the indices of the restriction conditions implied by the transition relation.

  For example, consider a transition τ with transition relation

$$\rho_\tau : \; \pi = \ell_1 \wedge x > 0 \wedge \pi' = \ell_2$$

  and abstraction and restriction instructions

$$\begin{array}{ll} \beta: & \{(\pi = \ell_1, \{\pi'\}), (\pi = \ell_4, \{x\})\} \\ \epsilon: & \{(\pi = \ell_1, \pi' = \ell_5)\} \end{array}$$

  Clearly ρ_τ implies the first abstraction condition in β, but not the second, and thus only π' is projected out, resulting in an abstracted transition τ_β with transition relation

$$\rho_{\tau_\beta} : \; \pi = \ell_1 \wedge x > 0$$

  Subsequent application of the restriction condition then results in the transition $\tau_{\beta\epsilon}$ with transition relation

$$\rho_{\tau_{\beta\epsilon}} : \; \pi = \ell_1 \wedge x > 0 \wedge \pi' = \ell_5$$

  Thus, the effect of β and ϵ on τ is to redirect the control flow of τ to a new location.

  Using the above notation, the new set of transitions $\mathcal{T}^*_i$ can now be given as

$$\mathcal{T}^*_i = \{\tau_{\beta_i \epsilon_i} \mid \tau \in \mathcal{T}_i\} \;\cup\; \mathcal{T}_{\alpha i}$$

4. ASPECT EXAMPLES

The aspect model introduced in the previous section is sufficiently expressive to represent many of the aspect constructs used in practical languages such as AspectJ [6]. In this section we present some examples of how these constructs can be represented in our model.

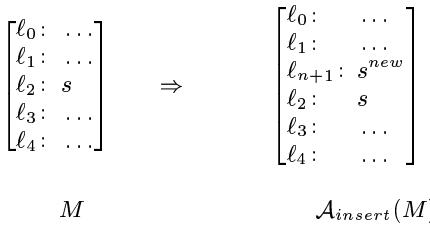


Figure 4: Inserting a statement before a given statement s

Insert Before

The most common aspect action is to introduce one or more statements before (or after) a given statement. Given a sequential program represented by the modular system $\mathcal{M} : \langle M \rangle$ with control variable π with range $\ell_0 \ldots \ell_n$, the following aspect inserts transition τ^{new} with (unspecified) transition relation ρ^{new} just before a given statement s.

$$\mathcal{A}_{insert} = \langle \emptyset, true, \alpha \rangle$$

The set of global variables is empty as no global state needs to be introduced, and there are no restrictions to the global initial condition. There is one aspect facet,

$$\alpha = \langle \emptyset, true, \{\tau^{new}\}, \beta, \epsilon \rangle$$

The set of local variables is empty and there is no change to the initial condition.[3] The facet transitions include the transition to be introduced, τ^{new}. The abstraction condition β needs to abstract the transition relation of the statement before s to enable redirection of control to the new transition:

$$\beta = \left\{ (\pi' = preloc(s), \{\pi'\}) \right\}$$

that is, variable π' is projected out from transition relations in which the control variable is set to the prelocation of statement s. Finally ϵ restricts the same transitions to direct control to a new program location ℓ_{n+1}, not occurring in the program

$$\epsilon = \left\{ (\pi' = preloc(s), \pi' = \ell_{n+1}) \right\}$$

The transition relation of the new transition now becomes

$$\rho_{\tau^{new}} : \rho^{new} \wedge \pi = \ell_{n+1} \wedge \pi' = preloc(s)$$

that is, the new transition is associated with the new program location and its post location is the prelocation of the given statement s.

Figure 4 shows the effect of the aspect in SPL program notation. Note that the above aspect is only one way of representing this construct. Others are possible and may be convenient in different situations (for example, if one does not want to modify the range of the control variable).

Logging

Given a system $\mathcal{M} : \langle M_1, M_2 \rangle$, the following aspect counts the changes to system variable x:

$$\mathcal{A}_{logging} = \langle \{N_x\}, N_x = 0, \alpha_1, \alpha_2 \rangle$$

[3]For simplicity we assume that s is not the first statement of M; if it is we do need to modify the initial condition.

It introduces a global aspect variable N_x, initialized to 0, to record the number of changes and includes two aspect facets $\alpha_{1,2}$:

$$\alpha_{1,2} : \{\emptyset, true, \emptyset, \emptyset, \epsilon\}$$

whose only significant component is the restriction instruction

$$\epsilon = \left\{ (true, (x' \neq x \rightarrow N'_x = N_x + 1) \wedge (x' = x \rightarrow N'_x = N_x)) \right\}$$

whose effect is to add the above conjunct to all transitions in $M_{1,2}$, incrementing N_x when x is modified, and leaving it unchanged otherwise.

An alternative restriction instruction is

$$\epsilon = \left\{ (x' \neq x, N'_x = N_x + 1), (x' = x, N'_x = N_x) \right\}$$

which, when used in actual program construction, would lead to a more efficient program. However, it relies on our ability to decide whether a transition relation implies the restriction conditions, which, in general, may not be decidable.

Adding synchronization

Consider a system $\mathcal{M} : \langle M_1, M_2 \rangle$ with two parallel processes with control variables $\pi_{1,2}$ ranging over program locations $\ell_0 \ldots \ell_n$ and $m_0 \ldots m_k$ respectively, that uses a resource S that requires mutually exclusive access. The following aspect adds protection of the resource using the *Bakery* synchronization mechanism shown in Section 2.

$$\mathcal{A}_{exclusion} = \langle \{y_1, y_2\}, y_1 = 0 \wedge y_2 = 0, \alpha_1, \alpha_2 \rangle$$

Two global variables are introduced, both initialized to 0. The two aspect facets are defined as follows:

$$\alpha_1 : \langle \emptyset, true, \{\tau_{11}, \tau_{12}, \tau_{13}\}, \beta_1, \epsilon_1 \rangle$$

with

$$\begin{array}{ll} \rho_{\tau_{11}} : & \pi_1 = \ell_{n+1} \wedge \pi'_1 = \ell_{n+2} \wedge y'_1 = y_2 + 1 \\ \rho_{\tau_{12}} : & \pi_1 = \ell_{n+2} \wedge \pi'_1 = preloc(S_1) \wedge (y_2 = 0 \vee y_1 \leq y_2) \\ \rho_{\tau_{13}} : & \pi_1 = \ell_{n+3} \wedge \pi'_1 \in postloc(S_1) \wedge y'_1 = 0 \end{array}$$

where $\ell_{n+1} \ldots \ell_{n+3}$ are new program locations not occurring in the program. The abstraction instruction is given by

$$\beta_1 = \left\{ (\pi_1 = preloc(S_1) \vee \pi'_1 = preloc(S_1), \{\pi'_1\}) \right\}$$

eliminating the control flow from the statement using the resource, and those preceding the statement that uses the resource. The restriction instruction restores the control flow:

$$\epsilon_1 = \left\{ \begin{array}{l} (\pi'_1 = preloc(S_1), \pi'_1 = \ell_{n+1}), \\ (\pi_1 = preloc(S_1) \wedge \pi'_1 \in postloc(S_1), \pi'_1 = \ell_{n+3}) \end{array} \right\}$$

Similarly,

$$\alpha_2 = \langle \emptyset, true, \{\tau_{21}, \tau_{22}, \tau_{23}\}, \beta_2, \epsilon_2 \rangle$$

with

$$\begin{array}{ll} \rho_{\tau_{21}} : & \pi_2 = m_{k+1} \wedge \pi'_2 = m_{k+2} \wedge y'_2 = y_1 + 1 \\ \rho_{\tau_{22}} : & \pi_2 = m_{k+2} \wedge \pi'_2 = preloc(S_2) \wedge (y_1 = 0 \vee y_2 < y_1) \\ \rho_{\tau_{23}} : & \pi_2 = m_{k+3} \wedge \pi'_2 \in postloc(S_2) \wedge y'_2 = 0 \end{array}$$

and similar definitions for β_2 and ϵ_2 as above.

Figure 5 shows the effect of the aspect in SPL program notation.

$$\begin{bmatrix} \ell_0\colon & \ldots \\ \ell_1\colon & S_1(uses\ S) \\ \ell_2\colon & \ldots \end{bmatrix} \quad || \quad \begin{bmatrix} m_0\colon & \ldots \\ m_1\colon & S_2(uses\ S) \\ m_2\colon & \ldots \end{bmatrix}$$

(a) $\mathcal{M} :< M_1, M_2 >$ using shared resource S

$$\begin{bmatrix} \textbf{shared}\ y_1, y_2 : \textbf{integer} \\ \ \textbf{where}\ y_1 = y_2 = 0 \\ \ell_0\colon \quad \ldots \\ \ell_{n+1}\colon\ y_1 := y_2 + 1 \\ \ell_{n+2}\colon\ \textbf{await}\ (y_2 = 0 \vee y_1 \leq y_2) \\ \ell_1\colon \quad S_1(uses\ S) \\ \ell_{n+3}\colon\ y_1 := 0 \\ \ell_2\colon \quad \ldots \end{bmatrix} \quad || \quad \begin{bmatrix} \textbf{shared}\ y_1, y_2 : \textbf{integer} \\ \ \textbf{where}\ y_1 = y_2 = 0 \\ m_0\colon \quad \ldots \\ m_{k+1}\colon\ y_2 := y_1 + 1 \\ m_{k+2}\colon\ \textbf{await}\ (y_1 = 0 \vee y_2 < y_1) \\ m_1\colon \quad S_2(uses\ S) \\ m_{k+3}\colon\ y_2 := 0 \\ m_2\colon \quad \ldots \end{bmatrix}$$

(b) $\mathcal{A}_{exclusion}(\mathcal{M})$ ensuring mutually exclusive access to S

Figure 5: Adding synchronization

$$\begin{bmatrix} \ell_0\colon\ i := 2 \\ \ell_1\colon\ \textbf{while}\ i < N\ \textbf{do} \\ \quad \begin{bmatrix} \ell_2\colon\ a[i] := f(a[i-1], a[i], a[i+1]) \\ \ell_3\colon\ i := i + 1 \end{bmatrix} \\ \ell_4\colon\ i := 1 \\ \ell_5\colon\ \textbf{while}\ i < N - 1\ \textbf{do} \\ \quad \begin{bmatrix} \ell_6\colon\ a[i] := g(a[i]) \\ \ell_7\colon\ i := i + 1 \end{bmatrix} \\ \ell_8\colon \end{bmatrix}$$

Figure 6: Program SERIES

$$\begin{bmatrix} \ell_0\colon\ i := 2 \\ \ell_1\colon\ \textbf{while}\ i < N\ \textbf{do} \\ \quad \begin{bmatrix} \ell_2\colon\ a[i] := f(a[i-1], a[i], a[i+1]) \\ \ell_9\colon i := i - 1 \\ \ell_6\colon a[i] := g(a[i]) \\ \ell_7\colon i := i + 1 \\ \ell_3\colon i := i + 1 \end{bmatrix} \\ \ell_8\colon \end{bmatrix}$$

Figure 7: $\mathcal{A}_{fusion}$(SERIES)

Loop Fusion

Our model of aspects is sufficiently expressive to represent loop fusion, a transformation often useful in image processing to optimize cache performance. Consider program SERIES shown in Figure 6. Assume the program has control variable π with range 0..8, such that $\pi = i$ when control is at location ℓ_i.

The following aspect merges the two loops starting at ℓ_1 and ℓ_5:

$$\mathcal{A}_{fusion} = \langle \emptyset, true, \alpha \rangle$$

with

$$\alpha = \langle \emptyset, true, \{\tau\}, \beta, \epsilon \rangle$$

with abstraction instruction

$$\beta = \left\{ (\pi = 1 \vee \pi = 2 \vee \pi = 7, \{\pi'\}) \right\}$$

and restriction instruction

$$\epsilon = \left\{ \begin{array}{l} (\pi = 1, (i < N \wedge \pi' = 2) \vee (i \geq N \wedge \pi' = 8)) \\ (\pi = 2, \pi' = 9) \\ (\pi = 7, \pi' = 3) \end{array} \right\}$$

The transition relation of the newly introduced transition is

$$\rho_\tau :\ \pi = 9\ \wedge\ \pi' = 6\ \wedge\ i' = i - 1$$

The result of applying $\mathcal{A}_{fusion}$ to program SERIES is shown in Figure 7.

5. ANALYSIS OF ASPECTS

Properties of reactive systems are often specified in some form of temporal logic. For example, the property that in all program behaviors the value of system variable x does not exceed 10 can be specified by the linear temporal logic (LTL) formula

$$\Box(x \leq 10)$$

where the $\Box$-operator means "always". Other temporal operators are provided that state that some condition p must eventually be fulfilled ($\Diamond p$), or that some condition p holds until another condition q holds ($p \mathcal{U} q$). The precise semantics of LTL can be found in [9].

Verifying that a system $\mathcal{S}$ satisfies a specification expressed as an LTL formula ϕ, usually written

$$\mathcal{S} \models \phi$$

consists of proving that all program behaviors of $\mathcal{S}$ satisfy ϕ, that is

$$\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\phi)$$

where $\mathcal{L}(\phi)$ is the set of all behaviors (infinite sequences of states) that satisfy ϕ.

Property Inheritance

The definition of a formal semantics of aspects now allows us to study questions of the type

$$\frac{\mathcal{M} \models \varphi, \ldots}{\mathcal{A}(\mathcal{M}) \models \varphi}$$

for some temporal property φ; that is, what restrictions on aspects guarantee preservation of system properties. For example, it is easy to see that for a given safety property φ

$$\frac{\mathcal{M} \models \varphi, \beta_{\mathcal{A}} = \emptyset, \mathcal{T}_{\mathcal{A}} = \emptyset}{\mathcal{A}(\mathcal{M}) \models \varphi}$$

where $\beta_{\mathcal{A}} = \emptyset$ stands for $\beta_i = \emptyset$ for all aspect fragments α_i in $\mathcal{A}$, and similarly, $\mathcal{T}_{\mathcal{A}} = \emptyset$ stands for $\mathcal{T}_i = \emptyset$ for all aspect fragments. Clearly all aspects in which no abstraction is applied, that is, no program behaviors are added to the system, and no new transitions are introduced, should preserve any safety property. Note that liveness properties may not be preserved as restriction may disable transitions necessary to achieve some goal.

As was suggested in an early work on superimposition [5], aspects can be classified by their inheritance properties. For example, one can distinguish *monitoring* aspects, which perform pure augmentation and therefore preserve all temporal properties, *regulatory* aspects, such as the one given above, which may turn unfair computations into fair ones, and thus cause liveness properties to be violated, and all other aspects, which cannot make any guarantees. We expect to be able to make a finer classification for this last class.

Property SuperImposition

Similarly, one may develop a notion of aspects satisfying certain properties and determining what restrictions on module systems are required to ensure that the resulting system satisfies the property, that is, under what conditions can we guarantee

$$\frac{\mathcal{A} \models \varphi, \ldots}{\mathcal{A}(\mathcal{M}) \models \varphi}$$

Aspect Interaction

The third question that can be studied in this framework is aspect interaction, that is, under what conditions does the following hold:

$$\mathcal{L}(\mathcal{A}_1(\mathcal{A}_2(\mathcal{M}))) \;=\; \mathcal{L}(\mathcal{A}_2(\mathcal{A}_1(\mathcal{M})))$$

That is, is the order of applying aspects significant?

6. DISCUSSION AND FUTURE WORK

Extensions

The model presented in this paper is the basic model upon which we intend to build several extensions. A first desirable extension is parameterization. In our current aspect description language, variables, predicates and transitions must be specified literally. It is a straightforward extension to the aspect description language to allow aspects to be parameterized by the components appearing in the aspect fragments, such as β, ϵ and $\mathcal{T}$. A more challenging extension is to add the capability of capture of context as, for example, provided in AspectJ.

The current model relies on rather coarse abstraction, potentially causing the set of behaviors to grow larger than necessary, thus making it harder to prove property preservation. We are currently investigating whether a more fine-grained abstraction such as assertion-based abstraction [2] would allow to increase the accuracy of the abstraction in a useful way.

For ease of exposition we have omitted fairness in this paper. To investigate inheritance and superimposition of liveness properties, fairness properties will have to be included.

Constructing systems

The model as presented cannot be used directly to construct new systems from aspects and base systems. Abstraction and restriction depend on our ability to decide the validity of the implications governing their applicability. In all examples shown these implications were decidable and easy to check. It would be interesting to determine which constructs can be expressed by decidable conditions. We are currently implementing a construction method in STeP (Stanford Temporal Prover) [1], using decision procedures to decide applicability, to experiment with the various constructs.

7. REFERENCES

[1] Bjørner, N. S., Browne, A., Colón, M., Finkbeiner, B., Manna, Z., Sipma, H. B., and Uribe, T. E. Verifying temporal properties of reactive systems: A STeP tutorial. *Formal Methods in System Design 16*, 3 (June 2000), 227–270.

[2] Colón, M. A., and Uribe, T. E. Generating finite-state abstractions of reactive systems using decision procedures. In *Proc. 10^{th} Intl. Conference on Computer Aided Verification* (July 1998), A. J. Hu and M. Y. Vardi, Eds., vol. 1427 of *LNCS*, Springer-Verlag, pp. 293–304.

[3] Finkbeiner, B., Manna, Z., and Sipma, H. B. Deductive verification of modular systems. In *Compositionality: The Significant Difference, COMPOS'97* (Dec. 1998), W.-P. de Roever, H. Langmaack, and A. Pnueli, Eds., vol. 1536 of *LNCS*, Springer-Verlag, pp. 239–275.

[4] Fisler, K., and Krishnamurthi, S. Modular verification of collaboration-based software design. In *International Conference on Foundations of Software Engineering* (2001).

[5] Katz, S. A superimposition control construct for distributed systems. *ACM Trans. Prog. Lang. Sys. 15*, 2 (April 1993), 337–356.

[6] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. Getting started with AspectJ. *Communications of the ACM 44*, 10 (October 2001), 59–65.

[7] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., and Irwin, J. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (1997), vol. 1241 of *LNCS*, Springer-Verlag.

[8] Lamport, L. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM 17*, 8 (1974), 435–455.

[9] Manna, Z., and Pnueli, A. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.