Separating and Managing Dependent Concerns

Separating Statechart Implementations from Base Code in an Embedded System

Gurcan Gulesir, Lodewijk Bergmans, Pascal Durr, Istvan Nagy
University of Twente
{gulesirg,bergmans,durr,nagyist}@cs.utwente.nl

ABSTRACT

Aspect-oriented programming offers the software engineer a powerful means for separating concerns that are tangled within a single module. Many articles have been written showing examples of how to modularize a certain concern as an aspect, separated from the concern(s) in the base code. The typical claim is that these modularized concerns can henceforth evolve independently. In this paper we are focusing on the cases where there is also a strong need for separating tangled concerns, to reduce development and maintenance cost. The paper is based on our experience in modularizing concerns in a large embedded software system. In this system, the concerns to be separated are tangled at a very fine-grained level, requiring us to introduce a novel (fine-grained) pointcut specification technique. In addition, the concerns to be separated are *not* independent, which requires a different way of managing changes in the software. In this paper, we explain the problems we encountered and present an aspect-oriented technique to separate and manage dependent concerns.

1. INTRODUCTION

A *Silicon wafer* is a small thin circular slice of pure silicon, which is used for producing integrated circuits. The Semiconductor Equipment and Materials Institute (SEMI) [3] introduces *standards* for silicon wafers, factory automation and system software. These standards offer consistent means to monitor behavior of *semiconductor manufacturing equipment* (SME). In addition, SEMI standards are the common glossary for communication between SME suppliers and customers. Meeting these standards is one of the primary requirements for SME suppliers. One of these suppliers is ASML, which is the world market leader in lithography systems. ASML's embedded software is the subject system from which we derive the examples in this paper.

Multiple SEMI standards define how SMEs should interact with their environment in a fully automated factory. In such a factory, an *operator* (human) communicates with the equipment via *host software*. Based on the SEMI standards, the host software monitors both the material flow and the operation of the SME. It also takes necessary actions in case of faults, errors, exceptions and emergencies. Typically, embedded SME software controls the operation of hardware components, to carry out various tasks. Therefore, a major part of the SEMI standards have to be implemented in software, to offer an interface for the host software. Lithography systems evolve at a pace that is predicted by Moore's law. As a result, both lithography systems and the SEMI standards change frequently to meet the evolving requirements of the domain. It is therefore essential for ASML that the software in the SMEs is sufficiently evolvable to keep up with all changing requirements.

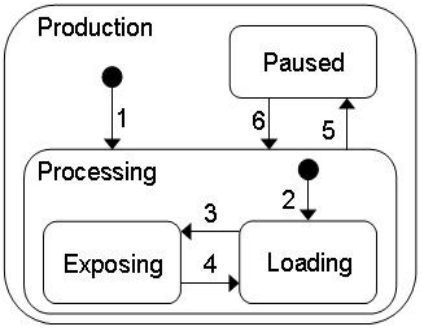

Figure 1: This statechart depicts some states of a wafer during production.

Statecharts [9] are used to model certain behavior of SME, and implement SEMI standards. Figure 1 shows a sample statechart[1]. Some of the states in the statecharts are SEMI states (i.e. specified in SEMI standards).

1.1 Current Implementation

The software system under consideration is a component based real-time embedded system, which has about 14 million lines of C code and contains approximately 200 components. Hundreds of statecharts are implemented within

[1]Due to the copyright restrictions of SEMI standards and the confidential nature of ASML source code, we are not allowed to expose them to a public audience. The statecharts and source code in this paper are fictitious but representative examples of the real ones, which we analyzed during our research.

this complex software. In this paper, we focus on one of the important components and one significant statechart.

Components contain many functions, which are the biggest units of abstraction in the C language. For simplicity in this paper, we ignore the notion of components. We distinguish two categories of statements: *core actions* and *transition triggers*. Core actions typically contribute to perform an equipment specific task. They change the internal state of the equipment by writing local or global variables, or calling other functions, or both. Therefore, upon execution of one or more core actions, state transitions in some statecharts might be necessary (i.e. multiple statecharts might be affected). A transition trigger is a statement in the source code that causes a state transition in a particular statechart's implementation.

In the following source code listing, two core actions: `CC_LoadWafer` and `CC_AlignWafer`, and transition trigger `CC_SetWaferState` of the `Wafer` statechart are visible. In fact, function `CC_ProcessWafer` contains more core actions and transition triggers, but they are not relevant for our discussion in this paper. I.e. each "..." in listing 1 is a placeholder for one or more consecutive (irrelevant) statements.

```
1  int CC_ProcessWafer(int wafer) {
2      ...
3      bool loaded = FALSE;
4      ...
5      CC_LoadWafer(wafer, &loaded);
6      ...
7      CC_AlignWafer(wafer);
8      ...
9      if(loaded == TRUE){
10         ...
11         CC_SetWaferState(wafer, EXPOSING);
12         ...
13     }
14     ...
15 }
```

Listing 1: Example source code, which shows the lines in the source code that are related to transition 3 in figure 1.

In this paper, we focus on the composition of statecharts with core actions. The actual implementation of statecharts is out of the scope of this paper. The dominant decomposition of the software is based on the structure of the core actions. As a result, the transition triggers are cross-cutting with respect to the core actions. This results in some complications, which we discuss in the next section.

1.2 Complications

As visible in listing 1, the code triggering a transition trigger (line 11) is tangled with core actions. This leads to some complications, as the software evolves. We show these complications in terms of the impact of some change scenarios:

Add If developers write new code (new core actions within an existing function or a new function from scratch), they have to take care of one of the following possible necessities regarding statecharts:
1. Performing a transition to a state at a proper place in the new code using one of existing outgoing transitions.
2. Creating new outgoing transitions for some states in the statechart implementation, and then considering 1.
3. Creating new states in the statechart implementation, and then considering 2.

Remove If developers remove existing code (existing core actions within an existing function or a complete existing function), they have to take care of one of the following possible necessities regarding statecharts:
1. Removing existing transition triggers in the code. This might lead to a deadlock, indefinite wait or some unreachable states within the statechart.
2. Removing existing outgoing transition for some states in the statechart implementation, and then considering 1.
3. Removing states in the statechart implementation, and then considering 2.

Refactor Refactoring existing code can be problematic, too. Similar to adding or removing code, developers have to take special care while changing the structure of the code. For instance, moving a particular core action to another location in the source code might require moving multiple transition triggers as well; otherwise, the overall behavior may not be preserved, and this would contradict the essence of refactoring [7].

1.3 Problem Statement and Goal

The way an SME operates must at all times be consistent with the way its associated host interprets the SME's operation. This means that the statecharts must always accurately reflect the executed core actions and state of the program. Therefore, for *some* new or modified core actions, specific statechart triggers need to be adapted. Due to the cross-cutting implementation of core actions and transition triggers, for *each* new or modified core action, developers need to carefully consider whether it is necessary to introduce or adapt transition triggers, even when it may turn out that the core action does not require any transition trigger at all. This results in unnecessary development and maintenance overhead as the software evolves; the goal of this paper is to reduce this overhead.

1.4 Approach

To solve the problem (see section 1.3), we propose an aspect-oriented mechanism that has two important features: First, it has a notification mechanism that observes core actions and notifies developers upon *the* changes that require adapting specific transition triggers. As a result, developers do not need to consider at all, whether it is necessary to adapt specific transition triggers when (re-)writing core actions. Second, it has some new constructs for expressing pointcuts, in such a way that it reduces the number of occasions where a developer needs to adopt transition triggers. As a result, the development and maintenance effort (see section 1.3) can be reduced.

Introduction of the aspect-oriented mechanism to the existing system follows a three stage approach. In the first stage (concern elimination), we eliminate transition triggers from the base code, so that we get pure core action code. In the second stage (aspect specification), we specify one aspect per statechart to modularize transition triggers. The first two stages can be thought of as the migration stages towards an aspect-oriented implementation. After the migration,

the development can continue in the new aspect-oriented setting, where the third stage (development) begins. Specific to this stage, the weaver takes the transition-trigger-free base code, and the aspects as the input, and produces source code with woven transaction triggers as output. The woven source code becomes the input of the C compiler. In this stage, a notification mechanism observes the system and automatically notifies developers when core actions are introduced or modified, and if these require adapting specific transition triggers.

Eliminating transition triggers and weaving aspects are in essence "source to source" code transformations. Control flow, data-flow, dependence graphs and abstract syntax trees etc. are commonly used representations of source code to perform such transformations. A control flow graph representation of each function is sufficient for performing the transformations in this paper. In this paper, we assume that we can create control flow graphs automatically from (normalized) source code, and vice versa. More concretely, we are using CodeSurfer [2] to create control flow graphs.

1.5 Outline

In section 2 we introduce some definitions. In section 3 we show how to eliminate transition triggers from the source code (stage 1), so that they are no more tangled with core actions. Following that, in section 4 we present how to specify transition triggers as aspects (stage 2). In section 5 we explain the continuous development stage (stage 3). Finally, we conclude in section 6 with the evaluation of our join point model in section 6.1 and the related work in section 6.2.

2. DEFINITIONS

We are going to use the following definitions and concepts in our algorithms and their explanations.

2.1 Control-Flow Graph

Below we give a formal specification of a control-flow graph of a normalized C function.

S is the statement set of a normalized C function F. S is divided into two subsets S_{cont} and S_{stmt} as follows: The statements belonging to S_{cont} are control statements (e.g. if, while, switch etc.) in S. The rest of the statements (core actions and transition triggers) in S belong to S_{stmt}. S_{trans} is the set of all transition triggers in S_{stmt}. A *control flow graph* G_s is defined as two finite sets E and V, over S. Elements of E are the *directed edges* of G_s. Elements of V are the *nodes* of G_s excluding the start node q and finish node f. V is further divided into two subsets: V_{cont} and V_{stmt} . b is a bijection defined as $b : S \rightarrow V$. For any $v \in V$ and $m \in S_{cont}$, $v \in V_{cont}$ holds if and only if $b(m) = v$. For any $v \in V$ and $m \in S_{stmt}$, $v \in V_{stmt}$ holds if and only if $b(m) = v$. V_{trans} is a subset of V_{stmt}. For any $v \in V_{stmt}$ and $m \in S_{trans}$, $v \in V_{trans}$ holds if and only if $b(m) = v$. $n \in S$ is a *possible successor* of $m \in S$ if and only if n might be executed right after m. There is a directed edge $e = (u, v)$ from $u \in V$ to $v \in V$ if and only if $b(m) = u$ and $b(n) = v$, where n is a possible successor of m. There is a directed edge $e = (q, u)$ from start node q to $u \in V$ if and only if $f(n) = u$, where n is not a possible successor of any $m \in S$. There is a directed edge $e = (u, f)$ from $u \in V$ to finish node f if and only if $b(n) = u$, where any $m \in S$ is not a possible successor of n.

2.2 Join Point Analysis

As we mentioned earlier, there are hundreds of statecharts for representing different SEMI standards, but worked on one of them in detail. For the statechart we worked on, there are 60 different places in 5 different software modules, where the transitions are triggered. As a result of our analysis, we concluded that a fine-grained join point model is required, which can enable developers to write pointcut specifications to select the exact join points of their interest. In this section, we present our join point analysis and in section 4 we provide a language for specifying aspects (i.e. pointcuts and advices) based on our analysis.

Within a function's execution context, the internal state[2] of a program can be represented by the values of some local variables or global variables, or both. We call such variables *state variables*. In listing 1, the boolean variable `loaded` is a local state variable in the execution context of `CC_ProcessWafer`. It determines the internal state of the program within the execution context of `CC_ProcessWafer`. A transition trigger typically occurs after the program reaches a particular internal state defined by the values of a set of related state variables within a function's execution context.

A *condition* is a conditional expression related to a transition trigger, in which related state variables are checked against desired values. In order to determine whether the program has reached an internal state related to a transition trigger, the related condition is checked at a specific lexical location in the source code. In listing 1, line 9 has the related condition check for the transition trigger in line 11. If the related condition is satisfied, then the transition trigger is performed. In line 11, `CC_SetWaferState` triggers the transition of the wafer statechart. As a result, the actual transition in the statechart's implementation changes the state of the statechart from the current state (i.e. whatever the state the statechart is in at that point of execution) to the `EXPOSING` state.

Related to a transition trigger, there are two types of core actions of interest: *Mandatory-Execute, Mandatory-Precede: MEMP* and *Optional-Execute, Mandatory-Precede: OEMP* actions. Both MEMP and OEMP actions might be nested (in multiple nestings of `if-then-else, while, for, switch` blocks) in different nesting levels than the related transition trigger. Execution of a transition trigger occurs if and only if the related MEMP actions are executed a priori. If ever the related OEMP actions are to be executed (i.e. they may not be executed due to a deeper nesting), then they must be executed before the transition trigger. The motivation for this classification is explained below.

The MEMP actions related to a transition trigger change the internal state of the program, defined by the values of a set of state variables related to the transition trigger. A condition related to the transition trigger must check these variables after the related MEMP actions and before the transition trigger are executed. In listing 1, `CC_LoadWafer` changes

[2]Note that this state is different than a state of a statechart.

the internal state of the program by modifying `loaded`. Since `loaded` is in the condition (line 9) related to the transition trigger (line 11), this distinguishes `CC_LoadWafer` as a MEMP action related to the transition trigger. Therefore, the transition trigger must be performed if and only if `CC_LoadWafer` is executed a priori. Note that there might be other core actions in lines 4, 6 and 8, but they do not modify `loaded`. Thus, they do not qualify as MEMP actions related to the transition trigger.

The OEMP actions related to a transition trigger either do not modify any variables or the variables they modify are not in the set of state variables related to the transition trigger. In listing 1, `CC_AlignWafer` does not modify any variables in the execution context of `CC_ProcessWafer`. It might modify some variables within its own execution context, but those variables are out of context for the transition trigger in line 11. However, the domain knowledge states the following rule: "A wafer can be aligned before exposure. During exposure (i.e. execution of the program after line 11), it is not possible to move a wafer.". Therefore, if ever a wafer is to be aligned, this must happen before exposure. This distinguishes `CC_AlignWafer` as an OEMP action related to the transition trigger. Note that there might be other core actions in lines 4,6 and 8, but they are not OEMP actions relative to transition trigger in line 11. Their existence before the transition trigger is either due to some other domain-specific fact which is not related to the transition trigger or for no specific reason (i.e. they could well be located after line 13, but this does not matter). Therefore, they do not qualify as OEMP actions related to the transition trigger in line 11.

2.3 Aspects

An aspect *aspect* is a tuple that consists of:

- *pointcuts*: A set of pointcut specifications, and
- *advices*: A set of advice specifications.

A pointcut specification *pointcut* is a quadruple that consists of:

- *within*: A *non-empty* set of functions,
- $MEMP$: A set of mandatory execute, mandatory precede actions,
- $OEMP$: A set of optional execute, mandatory precede actions, and
- *condition*: A conditional expression.

An advice *advice* is a tuple that consists of:

- *pointcutNames*: A set of pointcut names, and
- *statechartTrans*: A transition trigger.

3. STAGE 1: CONCERN ELIMINATION

We have already shown that transition triggers crosscut with respect to core actions. Therefore, we need to identify and eliminate transition triggers from core actions.

Figure 2 shows control-flow graphs of the source code in listing 1 before (figure 2a) and after (figure 2b) concern elimination. Note that declaration and assignment of the boolean variable `loaded` is separated in the control flow graphs. This is a typical side-effect of source code normalization.

Below we present our algorithm for eliminating transition

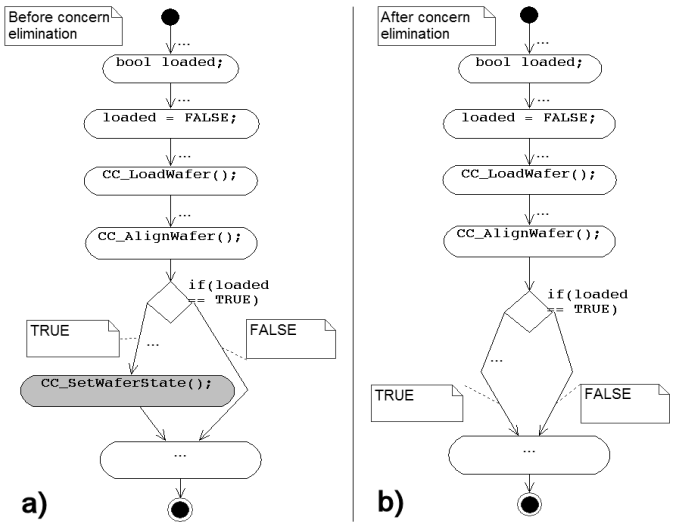

Figure 2: Control-flow graphs of the source code in listing 1 before (a) and after (b) concern elimination. The dark node is the eliminated node.

triggers from a control-flow graph. In our algorithms, we use the notation $G_s.V_{stmt}$ to explicitly denote that V_{stmt} belongs to G_s. We believe using such a notation makes reading easier, although this information is inferrable from the context of the algorithm and from the formal definitions in section 2.1.

algorithm EliminateConcern (G_s)
(1) **for** each $u \in G_s.V_{trans}$
(2) **for** each $v \in G_s.V$ and $v \neq u$
(3) **if** edge $(u, v) \in G_s.E$ **then**
(4) $G_s.E \leftarrow G_s.E \setminus \{(u, v)\}$
(5) $successor \leftarrow v$
(6) **for** each $v \in G_s.V$
(7) **if** edge $(v, u) \in G_s.E$ **then**
(8) $G_s.E \leftarrow G_s.E \setminus \{(v, u)\}$
(9) $G_s.E \leftarrow G_s.E \cup \{(v, successor)\}$
(10) $G_s.V \leftarrow G_s.V \setminus \{u\}$

This algorithm searches for every transition trigger node in a given control-flow graph G_s, and removes them. First, the algorithm searches for the possible successor of a particular transition trigger u (lines 2 and 3). There is always a unique possible successor of a transition trigger, because a transition trigger has a unique outgoing edge (i.e. transition trigger is not a control statement). Once the algorithm finds the successor, it removes the edge from the transition trigger to the successor (line 4). Second, the algorithm replaces every incoming edge to the transition trigger with a new incoming edge to the possible successor (lines 8 and 9). Finally, the algorithm removes the transition trigger from the control-flow graph (line 10). The resulting source code after concern elimination contains only core actions.

4. STAGE 2: ASPECT SPECIFICATION

In this section, we propose a template for a language, which is based on our join point analysis in section 2.2, to specify aspects. Using this language, developers can specify aspects to compose transition triggers with core actions. After we present the language through a template, we provide an example aspect specification for the transition trigger in listing

1 line 11. The language we propose in this section is by no means formal and complete. Here we only try to show some intuition and provide the general idea behind the language. Formal specification and implementation is our future work.

See section 2.3 for definitions of **bold** keywords. By definition, **within**[3] is obligatory, but **MEMP**, **OEMP** and **condition** are optional (i.e. can be empty). The *italic* words are identifiers. A *pointcutName* is a string to be specified by developers. A pointcut can have zero or more *arg*s which are specified in the same way as C function arguments. How these arguments are resolved is explained later in this section using the aspect example in listing 2. A *function* is defined as a combination of tokens and wildcards to match the signatures of the functions of interest. A *coreAction*, a *conditionalExpression* and a *transitionTrigger* are defined as combinations of tokens and wildcards to match the statements of interest.

aspect *aspectName*
{
  pointcut $pointcutName(arg_1, ..., arg_f)$:
    within$(function_1, ..., function_h)$,
    MEMP$(coreAction_1, ..., coreAction_i)$,
    OEMP$(coreAction_1, ..., coreAction_j)$,
    condition$(conditionalExpression)$;

  advice : $pointcutName_1(arg_1, ..., arg_k)$
      ... $pointcutName_l(...)$:
    $statechartTrans$;
}

In listing 2, there is an aspect specification for the transition trigger in listing 1, line 11.

```
1  pointcut exposingEventPointcut(int wfr) :
2    within(CC_ProcessWafer),
3    MEMP(CC_LoadWafer(wfr, &ldd)),
4    OEMP(CC_AlignWafer),
5    condition(ldd == TRUE);
6
7  advice : exposingEventPointcut(wfr) :
8    CC_SetState(wfr, EXPOSING);
9  }
```

Listing 2: Aspect specification for the transition trigger in listing 1, line 11.

The weaver resolves the pointcut variables `wfr` and `ldd` during weave-time. While traversing the control-flow graph of `CC_Process Wafer` (see figure 2b), it finds the node related to the function call `CC_Load Wafer`. At this point the weaver finds out the variables in the source code, and replaces the pointcut variables with them. In our case, the weaver replaces `wfr` with `wafer`, and `ldd` with `loaded` (see listing 1 for actual variables). Once the pointcut variables are resolved, the weaver is ready to weave inline. The advice is always an after advice. We explain the inline weaving in section 5.1.

5. STAGE 3: DEVELOPMENT

[3]In case only within is specified, then the advice is woven after the first node in the control-flow graph. Please follow the algorithm in section 5.1 for complete understanding.

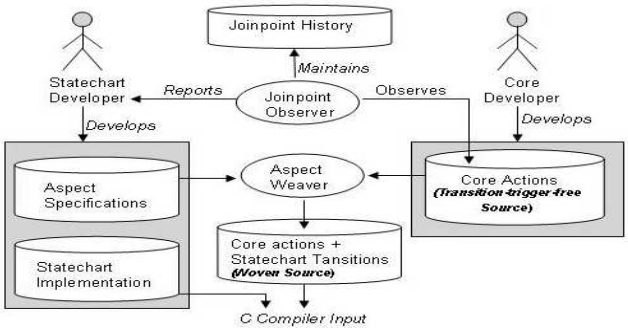

Figure 3: An aspect-oriented setting for development.

After concern elimination (stage 1) and aspect specification (stage 2) stages, the system has migrated to an aspect-oriented setting (figure 3). In this setting, we define two developer roles: *statechart developer* and *core developer*. Statechart developers implement and maintain aspects and statecharts. Core developers implement and maintain transition-trigger-free core actions. The weaver takes aspects and transition-trigger-free core actions, and weaves them. The woven source code and statechart implementation becomes the input of the C compiler. The join point observer is explained in section 5.2. In section 5.1, we explain the operation of the weaver.

5.1 Aspect Weaving

The aspect weaver takes the transition-trigger-free core actions (see figure 2b) and the aspect specifications for the transition triggers (see listing 2) as its input. And then, it weaves the transition triggers into the core actions to produce its output. The output of the weaver (see figure 4), which is a control-flow graph, is later transformed into the corresponding C source code. This source code is the input of the C compiler.

Below is the algorithm that describes how to weave transition triggers into the core actions. It may not be efficient, however providing an optimal algorithm is out of scope in this paper. The algorithm aims at finding the first node *join node* in the graph according to the constraints enforced by MEMP and OEMP actions. Once it finds the join node, it weaves the advice body and related code after it. The algorithms of *NodesAfter(v, G_s), FindJoinNode(counter, currentNode, result, G_s)* and *WeaveBeforeJoinNode(G_s, aspect, joinNode)* can be found in [1].

algorithm WeaveAspect (G_s, *aspect*)
(1) $result \leftarrow G_s.V$
(2) **for** each $coreAction \in aspect.MEMP \cup aspect.OEMP$
(3) Let $v \in G_s.V$ such that $b(coreAction) = v$
(4) $result \leftarrow result \cap NodesAfter(v, G_s)$
(5) $joinNode \leftarrow FindJoinNode(0, G_s.q, result, G_s)$
(6) $WeaveBeforeJoinNode(G_s, aspect, joinNode)$

In lines 2,3 and 4, the algorithm selects all nodes which come after and including all MEMP and OEMP actions with respect to the partial order of the control-flow graph. These nodes are stored in the node set named *result*. In figure 2b, such nodes are all the nodes below `CC_AlignWafer()`. At the end of the for loop, the set *result* contains all join node

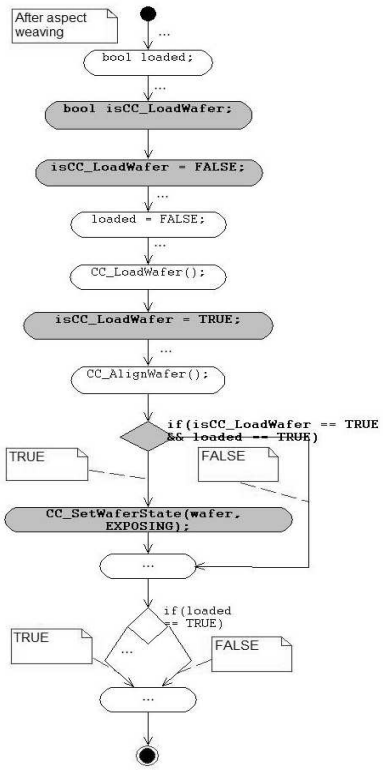

Figure 4: The control-flow graph after weaving the aspect in listing 2. Grey nodes are the woven nodes.

candidates. The algorithm in line 5 finds the first node in the result set with respect to the partial order imposed by the control-flow graph. That node becomes the join node. In figure 2b, join node is `CC_AlignWafer()`.

Finally, the algorithm in line 6 weaves the advice after the join node. It also weaves one boolean variable ("dirty bit") per MEMP action (e.g. `bool isCC_LoadWafer` in figure 2b). These boolean variables are also added to the condition check of the advice to make sure all MEMP actions are executed in runtime before the transition trigger is performed. Such boolean variables are not necessary for OEMP actions. Because, it is not mandatory that OEMP actions are executed. This is the only difference between MEMP and OEMP actions from weaving point-of-view.

5.2 Join Point Observer

The *join point history* (figure 3) contains one join point list for each pointcut specification for each compilation. At each compilation, the *join point observer* detects added and removed join points that are due to the changes in core actions, then it archives this information by inserting a new join point list per pointcut into the *join point history*. Subsequently, it informs statechart developers about these changes, so that they can consider adapting pointcut specifications.

An automated means of detecting and archiving the join point changes is crucial for communication between statechart and core developers. Without such a mechanism, it would be very difficult, if not impossible, to track join point modifications in the base code and then adopt related aspects.

6. CONCLUSION

6.1 Evaluation

In section 1.4 we claimed that an automatic notification mechanism would help reducing the development and maintenance cost. In section 5.2, we explained such a mechanism, which we call join point observer. The reduction in development and maintainence cost due to the join point observer is two-fold: First, statechart developers save reasoning effort related to *the* core action changes that neither add nor remove any join points. Note that this was not the case in the original (cross-cutting) setting (see listing 1). Second, if a change in the core actions result in addition or removal of one or more join points, then statechart developers can easily pinpoint these changes without manual searching. This provides statechart developers with faster means to reason about the join point changes and adapt pointcuts if necessary. Note that in the original (cross-cutting) setting (see listing 1), developers would have to manually browse the source code to pinpoint the altered transition triggers.

In section 1.4 we claimed that our aspect-oriented mechanism has new constructs for expressing pointcuts, in such a way that it reduces the number of occasions where a developer needs to adopt transition triggers. In section 4, we explained these constructs. Herewith we support this claim by evaluating the impact of a simple change scenario:

While we were analyzing the join points in section 2.2, we stated the following domain knowledge: "A wafer can be aligned before exposure. During exposure (i.e. execution of the program after line 11, listing 1), it is not possible to move a wafer.". Now assume that in this scenario, the alignment feature is modified, thus wafer alignment becomes possible only *before* loading wafer. Consequently, the domain knowledge evolves into: "A wafer can be aligned before loading. After a wafer is loaded (i.e. execution of the program after line 5, listing 1), it is not possible to move a wafer.". This requires a core developer to swap lines 5 and 7 in listing 1. Fortunately, any statechart developer neither needs to know these changes in the domain knowledge and the core actions, nor does he need to adopt the pointcut specification in listing 2. Because the statechart developer does not specify the order of execution of MEMP and OEMP actions in the pointcut specification. Therefore, the join point is flexible enough to handle this evolutionary change (i.e. the pointcut is not *fragile*[10] with respect to this specific change). As long as one of the MEMP or OEMP actions are not removed from the function specification in listing 1, the join points in that function specification will tolerate most of the core action changes. Note that (most of) the possible pointcuts specified using existing pointcut languages would be fragile, thus requiring adaption of the pointcut specification according to the core action change that is a consequence of the change scenario.

6.2 Related Work

In [6], we discuss a fine-grained C join point model that uses system dependence graphs, control flow graphs and abstract syntax trees as the program representations on which a weaver can operate. The join point model is capable of identifying any program point as a join point. We are going to build the join point model we have presented in this paper, on top of the fine-grained and powerful join point model

in [6].

In [10], the *fragile pointcut problem* is defined and a pointcut delta analysis is introduced to tackle the problem. Our join point observer in section 5.2 performs pointcut delta analysis as well. As an extra feature, the join point observer maintains the join point history for archiving purposes. We believe that archived the pointcut delta information will be useful for off-line analysis and visualization of how aspects have been interacting with the base system during software evolution.

AspectC [4] is an extension to C. It does not provide explicit support for identifying a set of statements in any order of execution as a join point. According to our join point analysis in section 2.2, AspectC is not expressive enough for the application in this paper. AspectC++ [11] has similar limitations as AspectC. In addition, although it can take C code as input source code, the output source code after weaving cannot be compiled by a C compiler.

Event based AOP [5] defines aspects in terms of events emitted during program execution. Pointcuts relate sequences of events. They are defined by event patterns that are matched during program execution (runtime). Once a pointcut has been matched, an associated action is executed. Although this seems a suitable join point model for the application in this paper, there are two limitations: First, runtime weaving usually has high performance penalty, and this is not acceptable for the high-performance applications like embedded SME software. Second, the pointcut matches join points if the events occur in the predefined sequence (i.e. according to the "rigid" pattern). This results in fragile pointcuts with respect to the changes such as the one we present in section 6.1.

In [8], an AspectJ implementation of standard Observer pattern is motivated and proposed to isolate notifications in aspects. If we consider statecharts as observers and core actions as the subject, then the transition triggers (notifications) crosscut the core actions (subject). From this perspective the aspect-oriented solution we provide in this paper seem to be similar to the one proposed in [8]. However, the pointcuts in AspectJ solution are fragile with respect to the changes such as the one we present in section 6.1. Moreover, the relationship between core actions and statecharts exhibit two well known applicability pitfalls of the observer pattern: First, no uniform way to notify statecharts (observers) exist. Second, statecharts (observers) cannot easily detect what has changed during execution of the core actions. The aspect-oriented solution we propose in this paper tackles these problems as well.

7. ACKNOWLEDGMENTS

We would like to thank Remco van Engelen and Ed de Gast from ASML for discussing the case study and for proofreading drafts of this paper, and all members of the Ideals project team, for their input about the topic. This work has been carried out as a part of the Ideals project under the management of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Senter program.

8. REFERENCES

[1] http://trese.cs.utwente.nl/~gulesirg/appendix.pdf.

[2] CodeSurfer. http://www.grammatech.com.

[3] SEMI: Semiconductor Equipment and Materials Institute. http://wps2a.semi.org/wps/portal.

[4] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 88–98. ACM Press, 2001.

[5] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 170–186. Springer-Verlag, 2001.

[6] P. Durr, L. Bergmans, and G. Gulesir. Towards an expressive and scalable join point model. *Submitted to FOAL '05: The 4th Workshop on Foundations of Aspect-Oriented Languages*, pages 200–209, March 2005.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Addison Wesley Professional, 1999.

[8] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173. ACM Press, 2002.

[9] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[10] C. Koppen and M. Stoerzer. PCDiff: Attacking the fragile pointcut problem. *1st European Interactive Workshop on Aspects in Software*, Sep 2004.

[11] O. Spinczyk, A. Gal, and W. Schroder-Preikschat. AspectC++: An aspect-oriented extension to the C++ programming language. In J. Noble and J. Potter, editors, *TOOLS Pacific 2002*, volume 10 of *Conferences in Research and Practice in Information Technology*, pages 53–60, Sydney, Australia, 2002. ACS.