Verification of Object Collaborations with Tracechecks

Thiago Tonelli Bartolomei

Generative Software Development Lab
University of Waterloo
Canada
ttonelli@uwaterloo.ca

Abstract

Tracematches is a mechanism to specify typestate properties over multiple collaborating objects. A tracematch lists (usually *incorrect*) behavioural patterns that trigger the execution of its body. In this paper we propose *tracechecks*, an approach in which the *correct* behavioural pattern is specified, and any unspecified pattern is considered incorrect, triggering the tracecheck body. Regular expressions traditionally have been used to specify behavioural patterns. However, we show that they are not appropriate to express partial orders or *correct* behaviour over multiple objects. We introduce *collaboration structures*, a convenient formalism to express such properties, and discuss its concrete semantics in detail. We also outline a possible approach to the static analysis of tracechecks and present a prototype implementation.

1. Introduction

During the execution of a program, objects collaborate by sending messages and ultimately changing their states. *Typestate* (Strom and Yemini 1986) can be used to express and verify temporal properties for an object. *Tracematches* (Allan et al. 2005) is a formalism that allows the specification of such properties. In comparison to previous approaches, such as (Fink et al. 2006), tracematches define a concrete language for expressing typestate properties and allow the specification of properties over groups of objects. Using tracematches, developers specify runtime events of interest, how objects relate to these events, the pattern of events that leads to some interesting state and a body of code to execute when the pattern is detected at runtime. Because traditionally tracematches have been used to perform runtime monitoring, usually the events lead to an *undesirable* (often *incorrect*) state, when an error is signalled or a countermeasure is taken.

A tracematch pattern specification must list *all* possible error sequences that should trigger the execution of its body. This is a limitation because developers may fail to specify some relevant error sequences, which are left unverified. To overcome this limitation we propose *tracechecks*, an approach in which developers specify the correct behavioural pattern for the objects of interest. Every detected sequence which is unspecified is considered incorrect and triggers the tracecheck body. We argue that tracechecks should be used jointly with tracematches to provide stronger verification guarantees, or stand-alone if the recovery code is uniform among the possible error sequences. In sec. 2 we give an overview of tracematches and discuss its limitations. We present tracechecks and examine usage scenarios in sec. 3.

Different formalisms can be used to express the correct behavioural pattern in tracechecks. In sec. 4 we claim that regular expressions, which are used by tracematches, are not appropriate to express *correct* behavioural patterns for multiple collaborating objects. Besides, they are not suitable to express partial orders among events. To tackle these shortcomings, we introduce *collaboration structures*, a formalism convenient to express partial orders between events that relate multiple objects. The exposition is accompanied by detailed discussions of their concrete semantics.

Like tracematches, tracechecks can be used for runtime monitoring. However, such approaches are more effective if their properties can be verified statically for all possible program traces. Static verification frees the system from the burden of runtime costs and increases the confidence in the correctness of program behaviour. In sec. 5 we sketch a possible approach to statically verify tracecheck specifications.

The contributions of this paper include:

- the concept of tracechecks to overcome some limitations found in tracematches (sec. 3).
- *collaboration structures*, a new formalism for the specification of partial order relationships among events that relate multiple objects (sec. 4).

Static Analysis for Software Engineering Fall, 2008.

- a sketch of how tracechecks could be verified statically (sec. 5).
- an initial prototype implementation for tracechecks and collaboration structures (sec. 6).

2. Tracematches

Tracematches (Allan et al. 2005) is a formalism that allows the specification of temporal properties over multiple collaborating objects. In comparison with other formalisms, it has the advantage of intuitive syntax and semantics, and of allowing the specification of properties over groups of objects. Consider the tracematch in listing 1 (adapted from (Allan et al. 2005)), whose goal is to verify the correct use of connection objects. It first declares a *free variable* `c` of type `Connection` (line 1). Then it declares named *symbols* (l.2-9), which specify with AspectJ (Kiczales et al. 2001) pointcuts the events of interest and how they bind the variable. The order of events that trigger the tracematch body (l.12) is defined by a regular expression (l.11) over the language of declared symbols, and can be represented by an automaton. The semantics of tracematches specify that one such automaton is created for each group of objects that get bound to the events. Also note that symbol `open` is not used in the pattern. Nevertheless, it is crucial to the proper working of the tracematch, as matching the `open` pattern discards the automaton. Only automatons that reach a final state execute the tracematch body.

```
1  tracematch ( Connection c ) {
2    sym open after :
3         call (* Connection.open()) && target(c);
4    sym close after :
5         call (* Connection.close ()) && target(c);
6    sym query before :
7         call (* Connection.query (..)) && target(c);
8    sym create after returning(c) :
9         call (Connection.new());
10
11   (create query) | (close query) {
12     c.open();
13   }
14 }
```

Listing 1. A tracematch checking connections.

Tracematches are useful for runtime monitoring because their sequences can be used to specify error patterns and the body can signal those errors or provide countermeasures. Because of this semantics, static analyses targeting tracematches (Allan et al. 2005) (Bodden et al. 2007) (Bodden et al. 2008) (Naeem and Lhoták 2008) traditionally focus on proving that the sequence of events specified by a tracematch never really occurs in a program and, if this cannot be proved, on how to minimize the runtime checks used to detect the sequence.

A limitation of tracematches is that *all* behavioural patterns of interest must be listed in the specification. In our example, the tracematch body will only execute if a sequence {*create, query*} or {*close, query*} occurs. Additional sequences that could be considered an error are not specified and will not trigger the tracematch body. The behavioural space of the objects can be seen as divided into correct and incorrect spaces. By listing error patterns, tracematches can be used to select particular sequences from the incorrect space, but anything left unspecified is considered correct. Hence, tracematches specifications leave a gap consisting of incorrect behaviour that is considered correct. In this work we propose *tracechecks*, which allow the specification of the correct behaviour of object collaborations. By considering unspecified behaviour as incorrect, tracechecks approach the behavioural space from the other direction and try to close the gap left by tracematches.

3. Tracechecks

Tracechecks allow the specification of the correct behaviour of object collaborations. A tracecheck's body is therefore executed when a certain object collaboration fails to obey its specification. For example, listing 2 shows a tracecheck that uses regular expressions to specify the correct behaviour of connections. In this example, the correct behaviour of a connection implies creating it and then *using* it zero or more times. In this context, *using* means opening, performing at least one query, and then closing the connection. Note that both error sequences from listing 1 are also captured here. Furthermore, a sequence such as opening and closing without querying is also an error in this example.

```
1 tracecheck ( Connection c ) {
2   ... // previous tracematch symbols
3   create (open query+ close)* {
4     throw new Exception(
5       "Connection " + c + " incorrectly used");
6   }
7 }
```

Listing 2. A tracecheck for connections.

A closer inspection of this tracecheck raises two important questions: *i)* what exactly constitutes incorrect behaviour? *ii)* what are the potential usage scenarios of tracechecks? These issues are addressed in the following.

3.1 Incorrect Behaviour

In the previous example a regular expression was used to specify the correct behaviour of the collaboration. Different formalisms could be used to specify the temporal relationships between the events. For example, in sec. 4 we propose a formalism that is convenient to specify partial orders between events. Yet, regardless of which formalism is used, a tracecheck can misbehave (and trigger its body) in two ways.

The first possible type of incorrect behaviour occurs when the sequence of events is not accepted by the specification. In our example, if the connection object is queried without being opened, the corresponding automaton cannot accept the `query` event and fails.

A second type of failure occurs if the sequence does not execute up to its end. For example, it is an error if the connection is not closed after it is queried. However, it is not clear when this property should be verified. In other words, for how long should the tracecheck wait for a `close` event after a `query` event, and when should the body be triggered? Ideally, the developer should be able to specify when to verify if the sequence is in an acceptable state to terminate. To this extent, we use the concept of *boundaries*.

A tracecheck specification should hold only within its boundary, which corresponds to its *initial* and *final* events. A tracecheck collaboration instance is created when an initial event occurs, and is terminated when it receives a final event. Final events must have a *single* multiplicity, because they must unambiguously determine the termination of a collaboration, and initial events must have a *mandatory* multiplicity (*single* or *some* - see sec. 4.3 for details on event multiplicities). For example, in listing 2 we can derive that `create` is an initial event. However, it is not clear which is the final event, as both `create` and `close` could terminate the tracecheck. In such cases, when the compiler cannot derive boundaries from the specification, it is necessary to declare an additional boundary symbol (sec. 4).

3.2 Usage Scenarios

The main target usage scenario for tracechecks, similarly to tracematches, is the verification of API usage patterns. Suppose that `Connection` in our examples is part of an API for database access. The tracecheck specifies exactly how this API class should be used at runtime. For a given application using the API, the compiler tries to statically prove that the tracecheck holds for all possible execution traces. If the compiler fails to prove the application correct (wrt. the tracecheck), it instruments the code to monitor the application at runtime.

But three characteristics of tracechecks distinguish their usage scenarios from tracematches. First, because their specifications can be stronger than tracematches, in the sense that the captured incorrect behaviour is a larger space, it can be used as an additional check, side by side with tracematches. For example, the tracecheck pattern specification in listing 2 could be added to the tracematch of listing 1, together with a boundary specification. When the compiler receives a tracematch with an additional check specification, it tries to prove it correct in the application and generate a certificate. If it fails, the tracecheck is simply ignored.

Second, tracechecks' boundaries can be helpful in delimiting the scope of the static analysis performed by the compiler. Fine grained boundaries can be used to verify properties in smaller parts of the system, which can increase the performance and precision of the static verification.

Finally, note that the tracecheck's body in listing 2 does not close the connection like the corresponding tracematch. The reason is that the tracecheck can fail in ways for which closing the connection is not the appropriate recovery. For example, if connections are opened and closed without being queried. Thus, we argue that tracecheck bodies should only be used when the recovery code is uniform among the possible error sequences, such as throwing and exception or logging the problem. Also, recovery code must be careful when using free variables, as tracechecks with multiple variables can trigger the body without having all variables bound.

4. Collaboration Structures

Tracechecks can use different formalisms to specify the temporal relationships between events. Regular expressions are convenient to express most types of allowed state transitions of a single object, as the connection example showed. However, regular expressions are not suited to express partial order relationships between events, as each possible order must be enumerated. Also, it is hard to reason on the semantics of a regular expression specification of *correct* behaviour when it involves multiple objects. To tackle these issues, we propose a formalism based on partial order specifications, which we call *collaboration structures*.

In its simplest form, a collaboration structure is a partial order between events, which implies that events can only appear in a certain order. Thus, collaboration structures are not appropriate to express specifications whereby an event can appear in different sequences. Informally, the collaboration structure semantics is that for an event to be accepted, all events that are *smaller* than it (wrt. the partial order) should have been accepted already by its collaboration. For instance, listing 3 contains a tracecheck using a collaboration structure to specify the allowed order of events occurring in a collaboration of menu and item objects (l.1). The pointcuts were omitted (l.3-9), but the corresponding comments describe the variables that are bound by the symbols. The partial order (l.11-12) specifies the order in which events should occur. A menu and an item can be created in any order (the coma represents a *logical and*), but before the item is added to the menu. Only after that can the menu be showed, a log must be registered and then the menu can be closed. Dispose of menu and item can also occur in any order, but after the menu is closed. Note that this partial order can have two final events. Therefore, a boundary was specified (l.2) to constraint the verification.

Sometimes it is necessary to specify a list of partial orders. Listing 4 is an extension to our previous example, that includes a panel object and three additional symbols. The partial orders (l.6-8) are separated by the semi-colon (l.7) and are assembled together by the compiler by symbol name. As the specification can become quite complex, we sug-

```
1  tracecheck ( Menu m, Item i ) {
2    boundary :  ...  // []
3    sym createMenu  ...  // [m]
4    sym createItem  ...  // [i]
5    sym addItemMenu ...  // [m, i]
6    sym showMenu ... // [m]
7    sym log  ...  // []
8    sym closeMenu  ...  // [m]
9    sym disposeMenu  ...  // [m]
10   sym disposeItem  ...  // [i]
11
12   createMenu,  createItem  < addItemMenu < showMenu <
13   log  < closeMenu < disposeMenu, disposeItem {  ...
14   }
15 }
```

Listing 3. A tracecheck for menus and items.

```
1  tracecheck ( Panel p,  Menu m, Item i ) {
2    // previous tracecheck symbols
3    sym createPanel  ...  // [p]
4    sym addMenuPanel ...  // [p, m]
5    sym disposePanel  ...  / [p]
6
7    createMenu,  createItem  < addItemMenu < showMenu <
8    log  < closeMenu < disposeMenu, disposeItem,  disposePanel ;
9    createMenu,  createPanel  < addMenuPanel < showMenu { ...
10   }
11 }
```

Listing 4. A tracecheck for panels, menus and items.

gest that IDEs provide a visual representation of the assembled collaboration structure. For listing 4 our implementation (sec. 6) produces the collaboration structure diagram in figure 1. The diagram is also useful to understand compiler errors. For example, if the boundary is not specified (listing 3) the compiler would produce an error because it would not be able to derive a well defined final event. As can be seen in the diagram, this specification contains three possible final events, which would be wrong without the boundary (on top). The developer can inspect the diagram to understand the error, and then either fix the partial order or declare a boundary.

4.1 Concrete Semantics

Before enriching the language to allow the specification of more complex behaviours, we shall clarify the concrete semantics of these simple collaboration structures. We start by classifying elements according to variable bindings. A symbol is called a *binding symbol* if it binds at least one variable, a *ground symbol* (Allan et al. 2005) if it does not bind any variable and a *grouping symbol* if it binds multiple variables. There must be a correspondence between initial and final symbols: if one initial symbol is ground then all boundary symbols must be ground. Runtime events matched by symbols are also classified into ground, binding and grouping events. A tracecheck whose boundary is ground is called a *ground tracecheck*. In the collaboration structure, an event has *parents* (the immediate previous events) and *children* (the immediate following events). The transitive closures are *ancestors* and *descendants*, respectively.

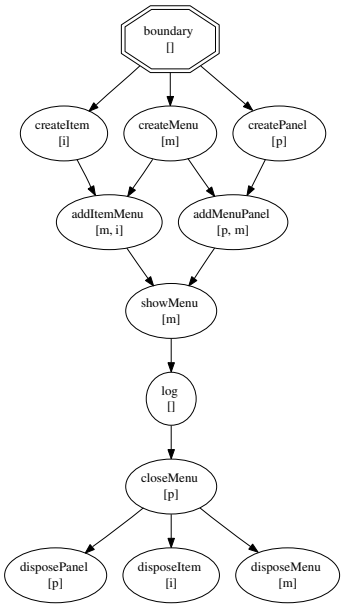

Figure 1. Graphical representation of listing 4.

The reason for the correspondence between boundary symbols will be clear after we discuss the concept of a *collaboration*. A collaboration is a specific instance of a tracecheck bound to some objects. It is said to be *complete* when all its free variables are bound. Every initial event instantiates a new collaboration, which initially can be incomplete. At any time, a runtime object can be bound to a certain variable only in a single collaboration (it is allowed for an object to take part in different collaborations of the same tracecheck if bound by different variables). A binding event is applied to the collaborations of the objects it binds, but ground events are applied to all live collaborations. A grouping event is applied to potentially many collaborations and, if the event is accepted, the collaborations are merged.

To better illustrate the rationale behind these rules we resort to our example in figure 1. Assume, however, that there is a single final event and that no boundary was specified. Then there are three possible initial events, all of them binding. If any of those events occur, a new collaboration is created, binding a single object. For example, the sequence $\{$*createItem*(i_1), *createMenu*(m_1), *createItem*(i_2), *createPanel*(p_1)$\}$ would create four collaborations, one binding each object. If the grouping event $\{$*addItemMenu*(i_1,

m_1)} follows, it will be applied to the collaborations corresponding to the i_1 and m_1 objects. Since both collaborations would be expecting this event, they accept it. The collaborations are then merged into one. When an {*addMenuPanel*(m_1, p_1)} event is detected, the collaboration corresponding to i_1 and m_1 is merged to p_1's, forming a complete collaboration. Suppose that it receives now a {*showPanel*(p_1)}. If the tracecheck then detects a {*log()*} event, it must verify that all live collaborations can receive this event (because it is ground). Note that the collaboration containing object i_2 is still alive, waiting for a menu object to collaborate. Our complete collaboration will proceed, as it is expecting the event, but the collaboration containing i_2 will fail, triggering the tracecheck body. Note also that this collaboration does not have all variables bounded to it, thus, tracecheck bodies must not assume that all free variables are bound. This process is repeated until the final event is reached (when the collaboration is terminated) or when an error occurs (an event is not accepted by the collaboration).

But consider the case of a ground tracecheck, which implies that the only initial event that can instantiate a collaboration is ground. In this case, the initial event instantiates a *ground collaboration*: a generic collaboration that binds no objects and simply collects ground events. For ground tracechecks, when an object is bound by an event and it has no attached collaborations, the ground collaboration is *cloned* and attributed to the object. The cloned collaboration will accept the event if it is a *start event*, i.e., a binding event whose ancestors are all ground. The cloned can then proceed with its bound objects until it terminates. However, it must be determined when the ground collaboration should be terminated. The logical approach is to terminate the ground collaboration when the final event occurs. But if the final event is binding, it would imply that only a single collaboration can get to the final event. Therefore, we enforce the rule that if a initial event is ground, so must the final event be.

For example, examine the exact case of figure 1, with a boundary declaration. Note that a boundary declaration is a convenience shortcut that specifies simultaneously the initial and final events: *before* and *after* its symbol matches, respectively. Developers could easily specify the corresponding symbols and modify the partial order to have them as the first and last events. In our example, the boundary is a ground symbol. Thus, when it is detected, a new ground collaboration is created. When a {*createItem*(i_1)} event occurs, i_1 does not have a collaboration. If the tracecheck were not ground, it would be an error. But in our case the ground collaboration is cloned and attributed to i_1. As this cloned collaboration has already seen the initial event, it can accept the `createItem` event. The process continues as previously described for non-ground tracechecks, until a final event is reached and the ground collaboration is terminated.

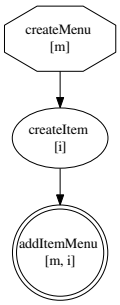

Figure 2. An incorrect collaboration structure.

4.2 Structure Rules and Thread Support

The concept of collaboration imposes additional rules to collaboration structure specifications: *i)* every variable bound by an event must be *live*, and *ii)* the effect of an event on its incoming collaborations must yield a single collaboration. The first rule guarantees that all variables used in an event have been bound before. A variable is said to be live if it is bound to a collaboration. Variables of initial and start events are always live because they start their own collaborations. Variables of grouping events are live if at least one variable is live in at least one parent. Variables of the remaining events are live if they are live in at least one parent. To clarify this first rule, consider the simple specification in figure 2, that defines a specific order in the creation of menus and items. Note that because there are no explicit boundary declaration, our implementation generates different shapes for initial and final events. A collaboration is created when {*createMenu*(m)} is received. The problem is that the {*createItem*(i)} event can never be attributed to this collaboration because there's no connection between the menu and item objects. Every such event would fail because there is no collaboration attached to i. In other words, the variable `i` is not live at symbol `createItem` (but `m` is live at `createMenu` as it is an initial symbol). Notice that objects are attributed to collaborations in initial and start events, but also by grouping events (if a variable of a grouping event is not live, provided at least another one is).

The second rule defines how collaborations are instantiated and merged. The effect of an event on collaborations depends on its type. Initial and start events simply instantiate collaborations with the objects they bind. Ground events do not affect collaborations wrt. instantiation and merging. Binding events with a single variable must verify that they receive a single collaboration. The compiler must check that by merging the incoming collaborations of all parents *by incoming variable*, a single collaboration is left. For example, the event `showMenu` in figure 1 receives a collaboration with *[m, i]* from `addItemMenu` and a collaboration *[p, m]* from `addMenuPanel`. By merging these collaborations through the variable m the compiler can verify that they are indeed the same collaboration. Grouping events can

merge collaborations, so they must verify that after merging there will be a single collaboration. The compiler first merges the incoming collaborations *by incoming variable* and then merges *by variables bound in the event*. For instance, the event `addMenuPanel` receives two collaborations from `createPanel` and `createMenu` which are merged by the variables m and p, bound to that event. However, if the structure had `showMenu` in the place of `addMenuPanel` (with `createPanel` and `createMenu` as parents) it would be an error, because `showMenu` binds only p. Thus, merging incoming p and m by variable would yield two collaborations, which is not allowed.

Finally, we must discuss threading issues. Tracechecks are by default *global*, in which case they intercept every occurrence of events in the system. However, they can be declared *perthread* so that only events occurring in a thread are considered. An indirect consequence of declaring tracechecks perthread is that explicit boundary declarations become control flow specifications. The reason is that explicit boundaries define that the initial and final events are the moments *before* and *after* matching its symbol. If a single thread is considered, it amounts to the control flow of the boundary event.

4.3 Event Multiplicity

The simple collaboration structures presented before specify events that must occur only once. But sometimes an event can be optional, or it can appear multiple times. To allow the specification of such cases, we introduce *event multiplicity*. By default events are *single*, meaning that they must occur once and only once. But they can be declared *lone* (zero or once, indicated by `?`), *some* (once or more, `+`) or *many* (zero or more, `*`). Events that are *lone* or *many* are also called *optional* because they may never occur in a collaboration's trace. *Single* and *some* events are also called *mandatory*. *Some* and *many* are also called *multiple*, because they can occur potentially multiple times.

In the tracecheck displayed in figure 1, for example, `addMenuPanel` could be made *some*, which specifies that the menu can be added multiple times to the panel, but at least once; `log` could be made *lone*, implying that it can happen or not, but at most once.

Event multiplicity affects some of our previous definitions. As mentioned before, final events must be *single* and initial events must be *mandatory*. The condition for initial events must be extended to start events of ground tracechecks since they represent in effect the initial events. Moreover, the collaboration structure rules must be adjusted to account for optional events. The rules guarantee that collaborations are properly instantiated and merged, and that variables bound by events always are live (bound to a collaboration). Since optional events can be skipped, their contributions to the collaboration's lifecycle must be conservatively estimated. To this end, we calculate a *reduced collaboration structure*, in which all optional events are removed,

```
1  tracecheck ( Panel p, Menu m, Item i+ ) {
2    // additional tracecheck symbols
3    sym showMenu? ... // [m]
4    sym showPanel ... // [p]
5
6    createMenu, createItem < addItemMenu <
7    (showMenu | showPanel) < log < closeMenu <
8    disposeMenu, disposeItem, disposePanel;
9    createMenu, createPanel < addMenuPanel < showMenu { ...
10   }
11 }
```

Listing 5. A tracecheck with alternative and optional events, and a multiple variable.

simulating the situation that they are skipped. When an optional event is removed, every parent is connected to all children. The collaboration structure rules are then applied to the reduced structure.

For example, if `addMenuPanel` were declared optional in figure 1, the reduced collaboration structure would connect its parents (`createMenu` and `createPanel`) to its child (`showMenu`). In this case, the collaboration structure rules would fail, as described in sec. 4.2, meaning that this event cannot be declared optional in this collaboration structure. Note that events such as `showMenu` or `log` could be optional, as the resulting reduced structure still complies with the rules. Listing 5 shows how to declare `showMenu` as *lone* (l.3).

4.4 Alternative Events

AspectJ allows the composition of pointcuts by logical conjunction and disjunction. A symbol could be specified with a disjunction of pointcuts to represent alternative events in the collaboration structure. However, AspectJ mandates that every pointcut in a composition binds the same variables. Hence, it is not possible to specify a symbol with alternative pointcuts that bind different variables. For instance, it is not possible to specify a symbol that matches if either a menu or a panel is shown.

Collaboration structures allow the specification of alternative events to overcome this problem. Alternative events are specified using the | operator, as shown in listing 5 (l.7). Alternatives are represented as nested events and only a single nesting level is allowed. The conjunction of `showMenu` and `showPanel`, for example, is treated as a single event in the collaboration structure, that can be matched by any of its constituents. When the compiler assembles the collaboration structure from the specification, merging of alternative events occur at the outer level. For example, it is not necessary to repeat the alternative (l.7) when referencing `showMenu` (l.9), because the compiler will merge these nodes in the collaboration structure.

In terms of live variables and event multiplicity, alternative events are also treated conservatively. An alternative event is optional if any of its constituents is optional (and the event is absent from the reduced collaboration structure). Also, if an alternative event is mandatory, the rules defined in sec. 4.2 must apply to every constituent separately, which simulates every possible alternative sequence.

4.5 Variable Multiplicity

The collaboration structures presented so far involve multiple variables, but each variable binds at most a single object. As tracechecks describe the correct behaviour of collaborations, allowing a single object per variable becomes too restrictive. For example, if the menu in our previous examples collaborates with multiple items, the tracecheck would trigger an error in the moment the menu object is grouped to a second item object (through an `addItemMenu` event). The error is triggered because the menu collaboration does not have space for an additional item.

Therefore, collaboration structures allow for variable multiplicity. Similarly to event multiplicity, tracecheck variables are by default *single*, but can be declared *lone*, *some* or *many*. However, only one variable can be *multiple*. The reason is that allowing several multiple variables would allow many-to-many relationships between objects, which would complicate the collaboration-based semantics. Listing 5 shows how variable multiplicity are declared (l.1).

Variable multiplicity defines how many objects can be bound to a variable, but also implicitly affects event multiplicities. If a variable is declared *lone*, every grouping event in which it participates becomes *implicitly lone*. When the compiler verifies the collaboration structure rules, it first considers the case in which an object is indeed bound to the variable (the standard case). Then it verifies a collaboration structure in which every event binding the *lone* variable is reduced. For example, if panel is declared *lone* in figure 1, the compiler would evaluate a reduced collaboration structure without events binding p (correct in this case).

If a variable is declared *some*, grouping events binding it become *implicitly some*. Because this multiplicity is mandatory, the compiler only verifies the standard case. At runtime, *implicitly some* events could occur many times. However, for a particular object bound to the variable, the regular event multiplicity is respected. The event is multiple only for the other variables bound by the event. For example, if item is declared *some* in figure 1, `addItemMenu` would be implicitly some. This means that for a particular item, it can occur only once (because the event is *single*), but for the menu object, this event can occur potentially many times, provided that different item objects participate.

A *many* variable affects the structure in both ways. First, because the variable is optional, the compiler must verify the collaboration structure in the absence of a bounded object. Second, because the variable is multiple, grouping events binding the variable are *implicitly many* and can occur multiple times to other participating objects.

5. Static Verification

We briefly sketch how an approach could be devised to statically verify tracechecks. As different formalisms can be used in tracecheck specifications, an analysis should take each one individually into consideration. Yet, here we focus on aspects common to any tracecheck formalism.

A static analysis for tracechecks can take advantage of the fact that only at initial (and start) events can collaborations be created, and that after created, they *must* execute until the final event. Therefore, an analysis can be staged in the following manner:

- for each symbol, the compiler computes its shadows in the system (the static footprint of a pointcut match). If there are no shadows for initial events, the compiler simply warns that the tracecheck cannot be applied.
- for each shadow of an initial symbol, the compiler computes the call graph starting at it and performs abstract interpretation. The abstraction depends on the formalism, but all it must do is verify if the events that are being encountered are accepted. If for every path a final event is definitely reached, the compiler proves the tracecheck correct. Otherwise, shadows are instrumented.

Note that the second step must be conservative in terms of possible control flow paths, threads, and especially aliasing. Also, we believe that a precise intra-procedural approach that conservatively treats inter-procedural relationships with summaries would provide the best trade-off between precision and performance.

6. Implementation

We implemented tracechecks and collaboration structures in a proof-of-concept prototype. The development occurred while the concepts presented in this paper were being unfolded. Thus, the current prototype does not include all features described previously. The prototype presently consists of the following modules.

Front-end

We extended the JastAdd (T. Ekman and G. Hedin 2007) tracematches front-end available in the AspectBench Compiler (Avgustinov et al. 2005). The front-end is able to parse tracecheck specifications, perform some semantic analyses and generate collaboration structure diagrams. The language accepted by the parser is different than the one presented in this paper, in particular:

- it is not possible to declare perthread tracechecks.
- event multiplicities are still described together with the partial order (instead of being specified in the symbols). Alternative events can also receive multiplicities, which

is not allowed anymore (although the constituents of an alternative event can have multiplicities).

- boundaries are specified with a *cflow* or *cflowbelow* kind, which was previously used to determine its type.
- pattern specifications can use regular expressions or collaboration structures. An `order:` token tells the parser that it is a collaboration structure.
- only the most simple semantic checks are implemented, such as unique initial and final event derivations.

The front-end implementation also includes build scripts, unit tests and an infrastructure to execute unit tests.

Runtime Library

We implemented a runtime library to support the execution of monitored applications. The compiler would then detect symbol shadows in the system and introduce code to notify the runtime library. The library keeps track of instantiation, merging and termination of collaborations, and executes the tracecheck body if an error is detected. The development of this library was essential in the formulation of the concrete semantics of tracechecks, presented in sec. 4.1.

7. Related Work

A number of approaches to specification and verification of program properties are related to our work. In this section we discuss a few that are most relevant.

Tracematches were proposed by (Allan et al. 2005) and are the main inspiration for our work. We have extensively compared tracechecks with tracematches, including the different usage scenarios (section 3).

J-LO (Bodden 2005) is a runtime verification framework in which temporal properties are specified using Linear Temporal Logic (LTL). J-LO is similar to tracematches in that it lists the patterns of interest, in this case using LTL, and the advice body is only executed under consistent variable bindings. Also, J-LO is a completely dynamic approach. To the best of our knowledge, there have been no attempts to verify J-LO specifications statically.

(Winskel 1987) introduced *event structures* as a formal notation to the specification of concurrent systems. In that context, event structures are used to specify consistent configurations of a concurrent system based on the events that it produces. The collaboration structures formalism presented here is different in that our focus is on object relationships and how they are grouped in collaborations.

8. Conclusions and Future Work

We presented *tracechecks*, an approach to verification of temporal properties over collaborating objects. Tracechecks specify the *correct* behavioural pattern for the objects and every unspecified behaviour is considered incorrect. We indicate that tracechecks can be used along tracematches to provide stronger verification guarantees, or stand-alone if the recovery code is uniform. We described how regular expressions can be used to specify tracecheck patterns, but argued that they are not appropriate to express partial orders or *correct* behaviour of multiple collaborating objects. We proposed *collaboration structures*, a new formalism to overcome these deficiencies, and discussed its semantics in detail. A possible approach to the static verification of tracechecks was outlined and our prototype implementation was presented.

We intend to extend our work in three directions. First, we intend to complete the implementation of the front-end and runtime library so that they represent the language and concepts presented in this paper. Also, we will implement the back-end, which will generate the code linking the symbol shadows to the runtime library.

Second, we will work on the further development and implementation of the static analysis, so that the compiler can minimize the runtime monitoring. Finally we intend to perform experiments in realistic case studies to evaluate tracechecks and the proposed static analysis.

References

Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to AspectJ. In *OOPSLA'05*, 2005.

Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An extensible AspectJ compiler. In *AOSD'05*. ACM, 2005.

Eric Bodden. J-LO - A tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University, 2005.

Eric Bodden, Laurie Hendren, and Ondřej Lhoták. A Staged Static Program Analysis to Improve the Performance of Runtime Monitoring. In *Proceedings of ECOOP*, pages 525–549, 2007.

Eric Bodden, Patrick Lam, and Laurie Hendren. Finding Programming Errors Earlier by Evaluating Runtime Monitors Ahead-of-Time. In *SIGSOFT '08/FSE-16*, pages 36–47. ACM, 2008.

Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. In *ISSTA '06*, pages 133–144. ACM, 2006.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *ECOOP*, pages 327–353. Springer, 2001.

Nomair A. Naeem and Ondřej Lhoták. Extending Typestate Analysis to Multiple Interacting Objects. Technical Report CS-2008-04, D.R.C. School of Computer Science, U. of Waterloo, 2008.

R E Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

T. Ekman and G. Hedin. The JastAdd Extensible Java Compiler. In *OOPSLA'07*, Montreal, Canada, 2007. ACM.

Glynn Winskel. Event Structures. In *Proceedings of an Advanced Course on Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986-Part II*, pages 325–392, 1987.