High-level Supervision of Program Execution Based on Formal Specification

Gergely Pintér, István Majzik
Department of Measurement and Information Systems
Budapest University of Technology and Economics
[pinterg, majzik]@mit.bme.hu

Abstract

This paper presents an approach for concurrently supervising the execution of applications specified by UML statecharts and a corresponding instrumentation scheme. The run-time verification is implemented by a statechart-level monitor while the instrumentation is based on Aspect-Oriented Programming.

1. Introduction

Our work in progress outlined in this paper aims at defining a framework for concurrent verification of object-oriented program execution with respect to both the intra-object *behavior* and the inter-object *communication*. The research is motivated by a current need for run-time verification against *formal models* as according to EN-50128 in case of software for railway control systems self-checking architectures are recommended. The model-based verification aims at detecting implementation faults (i.e. programming bugs, misunderstood specification etc.) resulting in a faulty series of actions or the delivery of incorrect results to the user. The requirements against the *verification mechanism* are as follows (Fig. 1).

- The reference information should be the *abstract specification* (not the implementation) seamlessly integrating to modern SW visualization methods e.g. specifying the behavior by statecharts and describing scenarios and communication by sequence diagrams or Live Sequence Charts (LSC) [1].
- It should be *configurable* to enable the focusing on key aspects of the specification. Promising formalisms according to this aspect are: temporal logic (TL) for liveliness and safety requirements, OCL (navigation) and LSCs for selecting scenarios [8].
- The verification (monitor) mechanism should be independent of the implementation i.e. *automatically generated from the specification* minimizing this way the possibility of common-mode faults.
- The *instrumentation* of the applications should be systematic, transparent, configurable and based on the abstract specification. Pattern-based approaches are viable candidates here e.g. the Aspect-Oriented Programming [4] paradigm.

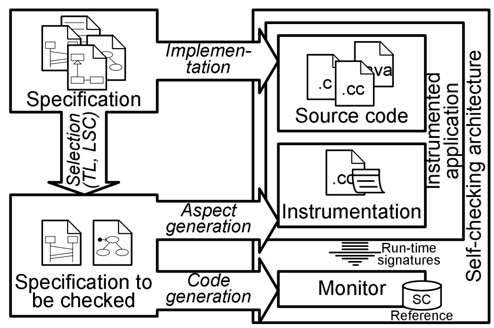

Figure 1. Obtaining the self-checking system

While designing the self-checking architecture outlined above two aspects should be distinguished. The purpose of *intra-object verification* is the detection of invalid trajectory in the state space resulting from implementation faults. The reference specifications can be UML statecharts. The communication is usually specified by sequence diagrams or LSCs. The purpose of *inter-object verification* is the detection of protocol violations, missed deadlines etc.

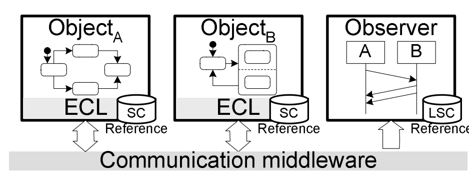

Figure 2: Architecture for verification

The unification of the two verification aspects implies an architecture shown in Fig. 2: the inter-object behavior is monitored by an embedded verification component taking the statechart specification as reference information while the communication is checked against the LSC specification by an observer component connected to the communication middleware. Based on the verification mechanisms an *error confinement layer* (ECL) can be inserted between the objects and the communication middleware enabling the implementation of fault-silent behavior or more advanced fault tolerance measures.

This paper focuses on the *intra-object verification*. The key contributions are the proposal for an *advanced monitoring scheme* for supervising the execution of statechart implementations and an *instrumentation method* enabling the investigation by the monitor without requiring manual modification of the application.

The paper is structured as follows. Sect. 2 introduces the intermediate reference formalism statecharts are transformed to, Sect. 3 proposes our monitoring scheme, Sect. 4 discusses the instrumentation method finally Sect. 5 outlines some experiment results, identifies the areas requiring further development and concludes the article.

2. Reference model

Since the dynamic semantics of UML statecharts is informal, our monitor aiming at intra-object verification uses the formally specified Extended Hierarchical Automata (EHA) [2] internally as reference model that statecharts are transformed to. Note that the internal representation of the monitor is invisible to the application designers, therefore the abstract models will be referred to as statecharts apart from this section.

An EHA consists of *sequential automata* (SA). A SA contains simple (non-composite) *states* and *transitions*. EHA states represent simple and composite states of the UML model. States can be *refined* to any number of SA. All automata refining a state are running concurrently (i.e. concurrent composite states are modeled by EHA states refined to several automata representing one region each).

Source and target states of an EHA transition are always in the same automaton. UML transitions connecting states at different hierarchy levels are represented by transitions with special guards and labels containing the original source and target states called *source restriction* and *target determination* respectively. At most one state in an automaton can be labeled as the initial state of the automaton building up the *initial state configuration* of the EHA.

The *operational semantics* (transition selection method) is expressed by a Kripke-structure in [2]. The execution of extended hierarchical automata is driven by *events*. A transition is *enabled* if its source state and all states in the source restriction set are active, the actual event satisfies the trigger and the guard is enabled. *Priority* of transition t_1 is higher than the priority of t_2 if the original source state of t_1 (indicated by the source restriction set) in the UML model is a directly or transitively nested substate of the original source of t_2. An enabled transition is *fireable* if there are no transitions enabled with higher priority. The (possibly several) transitions selected for firing are taken concurrently (i.e. in a non-deterministic order).

On taking a transition the source state is left *recursively* (with all active refinements) and the target state and all states in the target determination set are entered. The original definition of EHA does not consider the representation of actions to be performed when exiting (entering) states or assigned to transitions. According to the UML semantics the following requirements are to be introduced: (1) exit (entry) actions of states are to be performed according to the state hierarchy starting with the innermost (outermost) state and (2) the action associated to the transition is performed after the last exit and before the first state entry action.

3. Concurrent intra-object verification

There were several *watchdog processor* (WDP) schemes [3] (both in HW and SW [7]) proposed in the literature for detecting low-level control flow faults (i.e. deviation from the correct machine instruction sequence). The WDP obtains the *run-time information* by observing the CPU fetch cycles *(derived signatures)* or by processing signatures of execution explicitly sent by the observed application *(assigned signatures)*.

The run-time signatures are checked against the reference information typically stored as the *control-flow graph* (CFG) of the application. Although the CFGs were successfully applied for supervising the execution of relatively low-level programming constructs (functions, interrupt routines etc.) the formalisms lacks the capability of expressing event-driven *hierarchical* state-transition models and *concurrent execution*. While classical WDPs are successful in detecting effects of transient HW impairments, handling *SW faults* (programming bugs, misunderstood specification, etc.) has remained on open issue.

The *high-level, EHA-based watchdog* (EWD) proposed in this paper overcomes the weaknesses identified above by explicitly storing the EHA representation of the application statechart as *reference information* and maintaining a local observer of the state configuration of the supervised one. The *run-time information* sent by the application holds identifiers of states and transitions *(i.e. assigned signatures)*. The task of concurrent control-flow verification can be decomposed into two abstraction levels (contexts):

- The *EHA context* is responsible for monitoring the *initialization process* (i.e. exactly the states in the initial configuration are entered and the sequence of entry actions corresponds to the state hierarchy) and the *transition selection method* (i.e. the trigger event equals to the event received by the object, the source and source restriction states are active and priority relations are not violated).
- The *transition context* is responsible for monitoring the firing of a *single transition* i.e. the states exited (entered) by the application are really left (entered) by the transition, the sequence of exit, associated and entry actions is valid etc.

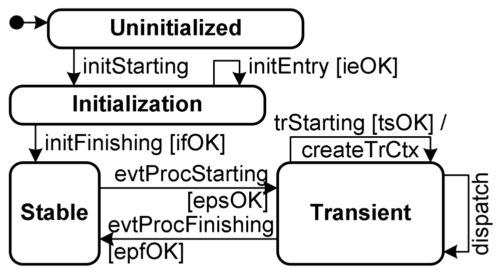

Figure 3: Operation of the EHA context

The implementation of EWD is based on the hierarchy and behavior of *contexts* identified above. The contexts are defined using protocol state machines (PSM) driven by messages sent by the application and the reference behavior is obtained from the UML statechart of the application. The PSM of the EHA context (Fig. 3) consists of four states corresponding to the lifecycle of the observed object: during construction *(Initialization state)* the object enters the states *(initEntry message)* belonging to the initial configuration. The start and finish of the initialization phase is indicated by messages *(initStarting* and *initFinishing)*. The consistent stable configurations of the observed object are represented by the *Stable* state. While processing an event the configuration is considered to be transient: transitions may be fired *(trStarting message)* and states are left and entered accordingly. Since several transitions may fire simultaneously according to our decomposition scheme a transition context is created for each running transition *(createTrCtx action)* and the messages related to them (e.g. state entry) are wrapped into messages sent to the EHA context and forwarded to the appropriate transition context *(dispatch message)*. The start and finish of event processing is indicated by messages *(evtProcStarting* and *evtProcFinishing)*.

The actual behavior supervision is implemented by *guard predicates* assigned to the transitions of the PSM. Entering a state during the initialization *(ieOK)* is valid if and only if (iff) (1) the state belongs to the initial configuration and (2) is currently inactive and (3) all the parent states were already entered. The initialization may be finished *(ifOK)* iff all states of the initial configuration were entered. A transition may be selected for firing *(tsOK)* iff (1) it is triggered by the currently processed event and (2) its source and source restriction states are active and (3) it is not disabled by an already started transition and (4) it does not disable an already started transition. The event processing may be finished *(epfOK)* iff all started transitions were successfully finished. Any messages not triggering a transition of the PSM are considered to be *protocol violations* (e.g. the reception of *initStarting* in *Stable* state). The PSMs and the guards discussed here are automatically generated on the basis of the reference statechart.

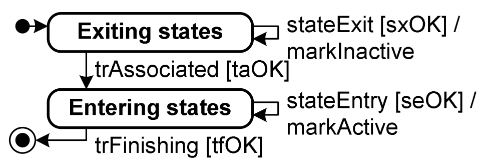

Figure 4: Operation of the transition context

The PSM of the transition context (Fig. 4) is driven by messages forwarded by the EHA context. Firing a transition involves three steps: (1) exiting the source state and all active states refining it *(Exiting states)*, (2) performing the action associated to the transition and (3) entering the target state and the ones in the target determination set *(Entering states)*. While leaving (entering) the source (target) states the application sends the *stateExit (stateEntry)* messages and the monitor updates its internal configuration observer accordingly *(markInactive* and *markActive)*. Before performing the action associated to the transition or finishing the transition the application sends *trAssociated* and *trFinishing* messages respectively.

A state may be exited *(sxOK)* iff (1) it is the source of the transition or one of its refinements and (2) it is active and (3) none of its refinements are active. The action associated to the transition may be performed *(taOK)* iff the source state and all of its active refinements have been left. A state may be entered *(seOK)* iff (1) it is the target of the transition or member of the target determination set and (2) it is inactive and (3) all of its parent states have already been entered. The transition may be finished *(tfOK)* iff the target and all states in the target determination set have been entered.

The watchdog discussed above was implemented as a stand-alone utility in ANSI C++ and successfully applied for detecting control flow faults in benchmark experiments. The prototype implementation is capable of supervising the execution of arbitrary number of objects by introducing a new topmost hierarchy level, the *application context* that is responsible for observing object construction and destruction (i.e. capable of detecting some types of memory leaks and corruptions) and dispatching the messages discussed above to EHA contexts. Note that the approach presented here supports the checking of *temporal logical requirements* by taking the accepting automaton of the TL specification as reference model.

4. Instrumentation

Since the concurrent verification scheme proposed above requires explicit transmission of assigned signatures to the EWD, observed applications have to be *instrumented* in two aspects: (1) the message processing interface of the EWD must be made accessible for the application and (2) message transmission routines have to be included at key control flow points identified above. The first aspect necessitates the extension of the *static data model* of the application i.e. the containment relation should be implemented. The second task requires the instrumentation *of the behavior*.

The approach followed by the application programmer for implementing state-based behavior has an important impact on the instrumentation method to be chosen. Instead of searching for a probably non-existing ultimate solution for instrumenting all possible statechart implementation techniques, we propose a *pattern-based approach* consisting of four steps:

- Identification of *extension points* in the *data model* where the static features for accessing the monitor (e.g. pointers etc.) are to be included.
- Identification of *key control points* in the *behavioral model* (e.g. methods recursively leaving the source state of a transition) where message transmission routines are to be included.
- Developing *instrumentation rules* that consist of (1) source code patterns matching one of the *instrumentation points* identified above and (2) source code *fragments to be applied* (included, substituted etc.) to matching points.
- Algorithmically *applying* the *instrumentation rules* to the source code of the application.

Since the implementation of statechart-based behavior is usually addressed by applying a design pattern proposed in the literature the process outlined above can be seen as developing *instrumentation patterns* for *implementation patterns*.

Figure 5 illustrates the *identification of extension points* according to our pattern-based approach in case of a simplified implementation pattern similar to [6] consisting of an abstract base class *(StatechartBase)* providing some fundamental facilities and a descendant class derived from it *(UserClass)* actually implementing the behavior. In this example the EWD is directly embedded in the application by adding a containment relation to the application class targeting the monitor instance with role "wd". One of the key methods of the pattern is the *fireTransition* function declared in the base class and implemented in the descendant. This function takes the necessary steps during the firing of a transition i.e. recursively leaves the source state, performs the action associated to the transition and enters the target states. Since the EWD requires the application to send a *trStarting* message before and a *trFinishing* message after firing a transition, the instrumentation inserts these actions in the behavioral model. The instrumentation-related elements (classes, actions etc.) are highlighted by grey surrounding.

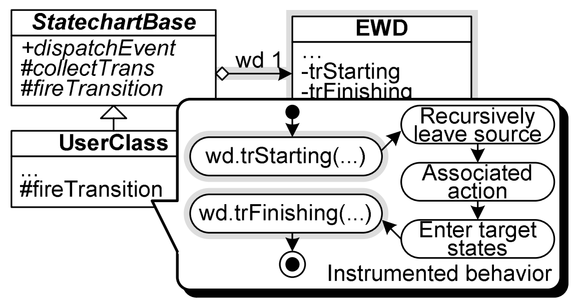

Figure 5: Instrumentation example

For implementing the pattern matching and introduction of static and dynamic instrumentation methods we propose the application of Aspect-Oriented Programming [4]. AOP aims at separation of concepts and enabling efficient maintenance of application code by distinguishing core and crosscutting concerns:

- *Core concerns* are the ones that belong to the primary purpose of the application. The design and implementation of core concerns is according to popular methodologies.
- *Crosscutting concerns* are features that should be implemented and integrated into the application but are difficult to design consistently together with the application since their purpose and related artifacts are independent of the primary purpose (e.g. resource accounting in a financial system).

Adding crosscutting concerns typically introduces several minor code fragments sporadically distributed throughout the entire source resulting in a non-maintainable, unreliable implementation. In order to overcome these drawbacks, AOP provides facilities for defining programming language-level (i.e. not primitive textual) patterns, so called *pointcuts* that can be automatically matched against application source. The matching regions are called *join points* while the modifications to be applied at join points are called *advices*. The class-like encapsulation of a set of pointcuts and advices is an *aspect*. Examples for possible patterns are method calls, object creation, accessing specific member variables (reading or writing), exception handling etc. The pattern matching can be recursive: pointcuts can be defined containing other pointcuts. AOP compilers are used to seamlessly *weave aspects* into the implementation of the primary application.

In our approach AOP is used for instrumentation of statechart implementations enabling the concurrent supervision by the monitor. In case of the example in Fig. 5 the *containment relation* between *UserClass* and *EWD* can be implemented e.g. in Java as adding a member variable of type *EWD* named *wd* to *UserClass*. The instrumentation of the *transition firing method* can be implemented as enclosing the original function body within calls informing the monitor about the start and finishing of the transition respectively. The aspect-oriented (AspectJ) implementation of the instrumentation is shown in Fig. 6. The first entry adds a new member variable to the abstract base class, the second one defines a pointcut as calls for the function *fireTransition* in classes derived from the *StatechartBase* class and finally the latest entry defines the instrumentation (advice) as discussed above (calling the appropriate methods of EWD before and after performing the original function body).

```
public aspect BehavioralMonitoring {
  // Add a  member variable to the base class
  protected EWD StatechartBase.wd;

  // Define a pattern (pointcut) called firingTransitionPattern
  // matching calls for fireTransition in derived classes
  pointcut firingTransitionPattern():
    call(StatechartBase+.fireTransition(Transition tr));

  // Define the advice (instrumentation) to be applied when
  // matching the previous pointcut
  around(): firingTransitionPattern() {
    wd.trStarting(tr);          // Send trStarting to the EWD
    proceed();                  // Perform original function body
    wd.trFinishing(tr);         // Send trFinishing to the EWD
  }
}
```

Figure 6: Instrumentation by AOP

This section has demonstrated our proposal for the instrumentation process required for concurrent supervision of statechart-based behavior. The instrumentation code is automatically generated on the basis of the pattern. In a more sophisticated implementation several instrumentation levels can be defined by corresponding aspect sets enabling the supervision of program execution at various abstraction levels in corresponding phases of the application life cycle (e.g. testing and operation).

5. Conclusion and future work

The key contributions of this paper were the proposal of a monitoring mechanism for run-time verification and a corresponding instrumentation scheme. The verification method enables the supervision of applications specified by UML statecharts. The instrumentation scheme seamlessly integrates to the pattern-based approach of modern software development methodologies and applies the emerging paradigm of AOP.

The stand-alone prototype implementation of the EWD was assessed by low-level fault injection into a statically initialized constant data structure encoding the behavior in an implementation pattern [5]. Although the bit-inversion faults do not exactly model software faults addressed by the watchdog, the fault detection ratio was remarkable (21.5% of detected faults was detected by the EWD only, 40% of injections resulted in HW exceptions, the remaining 18.5% were SW assertions). The full elaboration of the instrumentation pattern for a code generation scheme and the integration of the EWD into a benchmark application is subject of our current development. The next step of our research will be the assessment of the EWD and the LSC-based communication verification by source-code mutation-based fault injection and the integration of our run-time verification framework into modern component architecture.

References

[1] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. In Formal Methods in System Design, 2001.

[2] D. Latella, I. Majzik and M. Massink: Towards a Formal Operational Semantics of UML Statechart Diagrams. In Proc. FMOODS'99, pp 331-347, 1999.

[3] A. Mahmood and E. J. McCluskey: Concurrent Error Detection Using Watchdog Processors – A Survey. In: IEEE Transactions on Computers – 37(2), 1988.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier and J. Irwin. Aspect-Oriented Programming. In Proc. of ECOOP, Springer-Verlag 1997.

[5] G. Pintér and I. Majzik: Automatic Code Generation Based on Formally Analyzed UML Statecharts. In Proc. FORMS-2003, pp 45-52, 2003.

[6] M. Samek: Practical Statecharts in C/C++. Kansas (USA), CMP Books, 2002.

[7] I. Majzik, J. Jávorszky, A. Pataricza and E. Selényi. Concurrent Error Detection of Program Execution Based on Statechart Specification. Proc. EWDC-10, pp 181-185, 1999.

[8] S. Uchitel, J. Kramer and J. Magee. *Synthesis of Behavorial Models from Scenarios.* IEEE Transactions on Software Engineering. Volume 29, Number 2, February 2003