Applying the Composition Filter Model for Runtime Verification of Multiple-Language Software*

Somayeh Malakuti, Christoph Bockisch and Mehmet Aksit
Department of Computer Science, University of Twente
P.O. Box 217 7500 AE Enschede, The Netherlands
{s.malakuti, c.m.bockisch, m.aksit}@ewi.utwente.nl

Abstract— **Complex software, especially the embedded one, is composed of multiple collaborating subsystems that are possibly developed in multiple languages. To verify the behavior of such software, a run-time verification system must deal with multiple-language environments both in its specifications and in its generated runtime verification modules. In this paper, we present the E-Chaser runtime verification system, whose front-end provides language-independent specifications, and whose back-end provides an extendable toolset with new implementation languages. E-Chaser is built based on the Composition Filter Model and extends it with the notion of synchronization messages and synchronization filters to verify the synchronization properties of multiple subsystems. The first prototype of E-Chaser was successfully used to verify various properties.**

Runtime Verification; Mutliple-Language Software; Composition Filter Model

I. INTRODUCTION

Reliability is the ability of a software system to perform its required functions under stated conditions for a specified period [1]. It can be attained via different techniques among which we are interested in applying the runtime verification technique [2] to ensure functional correctness of the software. Runtime verification is the process of checking whether the active execution trace of software adheres to given properties, defined as **specification** of the software. In contrast to other verification techniques (e.g. model checking, or testing) which aim at checking all possible execution traces of the software, runtime verification reduces the verification scope to one execution of the software; this increases the accuracy of the verification, especially for dynamic properties of the software. While static verification techniques can only be used to remove the faults during the development phase [1], the results of runtime verification can additionally be used, for example, to tolerate the failures.

The **modular programming** technique increases the extent to which software is composed from separate modules. One example of modularized software is called **multiple-language** software, which is composed of subsystems implemented in different languages. A popular example of such software are embedded systems in which some subsystems must provide lower-level functionalities that are related to hardware, operating system, drivers and so on; whereas, the others must provide higher-level functionalities such as the user-interface. According to the functionality of each subsystem, different implementation languages may be employed. In our research group, we deal with various examples of such embedded multiple-language software. Philips MRI software, ASML wafer scanner and Océ printer software are three examples [20].

There are some attempts [3, 4] to support language-independent runtime verification systems which may potentially be used for verification of multiple-language software. However, either their implementation is only available for one language, or they cannot verify the synchronization properties of multiple subsystems or they only support software developed using specific infrastructure such as CORBA [5]. Therefore, when we utilize these verification systems, we must sacrifice the desired accuracy of the verification by abandoning the verification of subsystems developed in unsupported programming languages or using unsupported infrastructures.

In the literature, several runtime verification systems have been developed [6-10], which can be applied to software developed in one specific programming language (e.g. Java). To apply these verification systems to multiple-language software, a developer has to utilize separate verification systems to verify the subsystems of the software individually. Employing separate verification systems has two drawbacks; first, it increases the human effort for verification of the software because the developer has to learn multiple verification systems. Second, it decreases the accuracy of the verification because it is not possible to verify synchronization properties across multiple subsystems.

In this paper, we introduce the E-Chaser runtime verification system for multiple-language software. Our focus in E-Chaser is the detection of failures; however, since E-Chaser allows the specification of arbitrary actions to be executed as results of the detection, it can also be applied, e.g. to achieve fault tolerance.

The verification of the multiple-language software is facilitated in E-Chaser by two means. First, E-Chaser in its

* This work has been carried out as part of the TRADER project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Bsik program.

978-0-7695-3878-5/09 $26.00 © 2009 IEEE
DOI 10.1109/ISSRE.2009.12

front-end provides language-independent specifications, which allow a developer to define properties of subsystems regardless of their implementation language. Second, in its back-end it provides a toolset extendable with new implementation languages, which allows generation of the runtime verification modules for various implementation languages. Currently, support for Java, C and .Net languages is provided.

The two mentioned features of E-Chaser result in reducing the developer effort of applying E-Chaser to multiple-language software because s/he only needs to deal with one verification system. E-Chaser increases the accuracy of the verification in two ways. First, no properties of the software are excluded from the verification due to unsupported implementation languages; and second the individual properties of each subsystem as well as the synchronization properties of multiple subsystems can be specified and verified.

E-Chaser is designed based on the concept of the Composition Filter Model [11]. In the Composition Filter Model, the messages (e.g. method calls or events) which are exchanged between objects are filtered for different purposes such as verification. In this manner, E-Chaser provides a runtime-verification filter that is used to verify a sequence of messages expressed as a regular expression. Since in the current Composition Filter Model, filters can only be used to verify properties of individual subsystems, E-Chaser extends it with the notion of **synchronization messages** and **synchronization filters**, to support verification of the synchronization properties of multiple subsystems. The specification of software is composed of specifications of individual subsystems plus specifications of synchronization among them.

E-Chaser utilizes and extends Compose*, which is a compiler for the Composition Filter Model, to generate executable verification code from the composition filters.

The rest of this paper is organized as follows. In section II we provide background of runtime verification. In section III, our running example and the problems of existing runtime verification systems are explained. Section IV provides an overview of the E-Chaser runtime verification system along with the Composition Filter Model. Section V provides detail information about the specification language of E-Chaser, where section VI provides detail information about its toolset. In section VII we discuss the operational semantics of E-Chaser; and in section VIII we discuss the applicability of E-Chaser and evaluate E-Chaser with some quality attributes. Finally, section IX discusses the conclusion and future work.

II. BACKGROUND: RUNTIME VERIFICATION OF FUNCTIONAL PROPERTIES

In the runtime verification of functional properties, typically, the occurrence of certain events is checked during software execution against the specified properties. Most of the existing runtime verification systems adopt two-layer architecture similar to the one shown in Fig.1.

In the specification layer, there are three sub-specifications called *Events*, *Properties* and *Actions*. Here, we define an event as the execution of a statement or a group of statements in the software. The granularity of a statement depends on the implementation language and/or the execution environment of the language. *Events* specify the statements of interests that must be verified. *Properties* are logical predicates over the specified events, which are specified in formalisms such as regular expressions. *Actions* specify pieces of functionality that must be executed when the evaluation of a specified property fails at runtime.

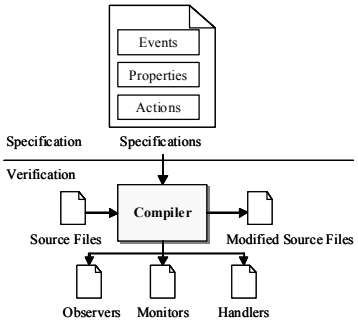

Figure 1. A typical architecture of runtime verification systems

In the verification layer, *Compiler* creates code for the runtime verification modules *Observers*, *Monitors* and *Handlers* from the specifications. *Observers*, which are generated from *Events*, are responsible of notifying *Monitors* about the occurrence of the specified events. *Monitors*, which are generated from *Properties*, are in charge of verifying the occurred events against the logical predicates and depending on the result of verification they call the corresponding handlers, if there are any. Likewise, *Handlers* are in charge of executing the functionality specified in *Actions*.

III. PROBLEM STATEMENT

In sub-section *A*, we will present an example application whose runtime behavior is to be verified. In sub-section *B*, this example is used to illustrate the problems addressed in the paper.

A. Running Example

The example software, whose runtime behavior is to be verified, is a media player composed of two subsystems: aTunes [12] and MPlayer [13]. They are implemented in two different languages, and are executed as separate processes. aTunes provides a user interface through which the end-user can enter commands. MPlayer implements the functionality for handling the commands and playing media files.

Fig.2 schematically shows the interactions between aTunes and MPlayer in handling the end user's request of increasing the volume of the currently played media. The end user's choice results in invoking the function *VolumeUp* on the module *GUI*, which updates the user interface by invoking the function *UpdateGUI* and invokes the function *WriteCommand* to send the request to *MPlayer*. The function *WriteCommand* passes the argument *VolumeUp* to the module *STDOUT*, which is part of the operating system's inter-process communication mechanism and sends this argument to the module *STDIN* of MPlayer. Next, the module *Core* reads the command

VolumeUp via the function *Read* and invokes the function *SetVolume* to change the output volume level. In this example, for the sake of brevity we removed the details of increasing the volume in the audio driver and the operating system level; however, Fig.2 could be extended with the sequences of messages which occur in these parts.

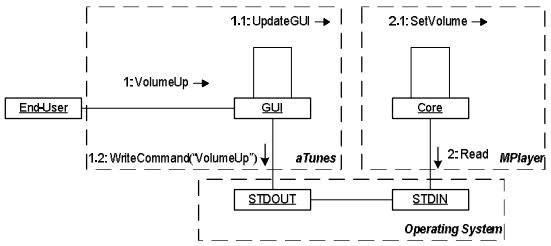

Figure 2. Interactions between aTunes and MPlayer

B. Shortcommings of Current Runtime Verification Systems

Let us assume we want to verify at runtime that the request to increase the volume by the end-user results in the increase of the output volume[1]. Although this looks like a rather easy task to do at first glance, there are however a number of challenges one has to overcome. The important challenge for us is that, the implementation depicted in Fig.2 is composed of modules implemented in two different languages.

In this section, we will discuss two possibilities: (a) using a single verification system for both language environments; and (b) using a different verification system for each language environment.

We will now elaborate on using a single verification system. In this case, the verification system must provide a specification language that is sufficiently abstract to express runtime behavior of software implemented in two different languages; this concerns the specification layer in Fig.1. In addition, the verification layer must be able to operate in two language-runtime environments.

In the literature, there are only a few attempts to make runtime verification systems language-independent. In MaC [3], for example, properties are defined in a language independent way in formalism similar to linear temporal logic (LTL). However, specifications of events refer to the constructs of the Java language. Although triggering of actions can be considered as language-independent, in the current version of MaC, actions can only be implemented in the Java language. At the time of writing this paper, the verification layer is only provided for Java although implementation efforts are reported for the C language as well.

In MOTEL [4] events are defined at the level of CORBA events, based on CORBA IDL. Therefore, the finer grained events, which are at the level of the programming languages, cannot be verified by MOTEL.

[1] We will assume that the sound driver and its usage are implemented correctly and only verify that the setVolume method is executed properly.

There are other runtime verification systems [14, 15] which focus purely on the specification logic and a monitoring engine for this logic, ignoring where events come from. However, the implementations of these tools are not publically available to be used for our evaluation.

Since currently there is no single verification system that supports fine-grained verification of multiple-language software, let us now elaborate on using a different verification system for each language environment. For example, aTunes is verified using the Java-based runtime verification system JavaMOP [6], whereas the C language-based MPlayer is verified using RMOR [7]. To further evaluate these systems, we will now specify our example case shown in Fig.2 using JavaMOP and RMOR. Fig.3 shows a specification of the aTunes events (see Fig.2) using the JavaMOP specification language, which is very close to the AspectJ language [16]; for brevity, we omit some implementation details such as the AspectJ specific declarations. Line 1 in Fig.3 specifies that the event *startVolumeUp* occurs before each invocation of the method *VolumeUp*. Likewise, lines 2 and 3 specify that the events *startUpdateGUI* and *startWriteCommand* occur before each invocation of the methods *UpdateGUI* and *WriteCommand*, respectively. Using the extended regular expression (ERE) language, line 4 specifies that the events *startVolumeUp, startUpdateGUI*, and *startWriteCommand* must follow each other subsequently for zero or more times. Line 5 specifies a Java statement that has to be executed when the events do not occur in the specified order.

```
1. event startVolumeUp before (): call ( * VolumeUp (..)){}
2. event startUpdateGUI before (): call( * UpdateGUI (..)){}
3. event startWriteCommand before (): call( * WriteCommand
(..)){}
4. ERE: ((startVolumeUp startUpdateGUI startWriteCommand)*)
5. @violation {System.out.println("Violation");}
```

Figure 3. A specification of aTunes in JavaMOP

MPlayer events (see Fig.2) are defined in Fig.4 using the specification language RMOR. Lines 1 and 2 specify that the events *startRead* and *startSetVolume* occur before each invocation of the functions *Read* and *SetVolume*, respectively, issued by a function implemented in the file *core.c*. Lines 3 to 5 specify the behavior of MPlayer as a state machine [17]. Line 3 specifies that *Retrieving* is the initial state, and upon the occurrence of the event *startRead*, the state machine enters the *Retrieved* state. Line 4 specifies that upon the occurrence of the event *startSetVolume*, a transition is performed from the *Retrieved* state to the *Finished* state, and line 5 defines the *Finished* state. For simplicity, not all details are shown here. RMOR provides a callback handler function, in the C language, which will be called for each detected violation.

```
1. event startRead = before call(core.c:Read);
2. event startSetVolume = before call(core.c:SetVolume);
3. initial state Retrieving {when startRead ->Retrieved;}
4. … state Retrieved {when startSetVolume -> Finished;}
5. … state Finished {…}
```

Figure 4. A specification of MPlayer in RMOR

In this approach, the Java-based and C-based subsystems are verified using JavaMOP and RMOR, respectively. There

are however at least two properties that cannot be verified adequately. Firstly, this approach does not guarantee that the sequence of events in aTunes is followed by the specified sequence of events for MPlayer. Therefore, an additional verification mechanism must be defined to check this dependency. Secondly, the inter-process communication between the two processes must be correct as well; otherwise, the two processes may not synchronize correctly.

One may try to address the dependency and inter-process synchronization problems by defining dedicated code that monitors the communication between aTunes and MPlayer. Defining such code for each different kind of inter-process communication, however, is a costly and error-prone task. It is therefore preferable to use a general-purpose inter-process verification system instead. This means using three different verification systems for a single thread of execution, which further increases the human effort to learn and apply the runtime verification systems.

IV. OVERVIEW OF E-CHASER

Instead of adopting various runtime verification systems and/or providing dedicated verification mechanisms, using a single runtime verification system for all language environments plus inter-process synchronization seems to be preferable. Therefore, we developed the E-Chaser runtime verification system for multiple-language software. E-Chaser facilitates the verification of multiple-language software in two ways. First, it provides language-independent specifications, which allow a developer to define properties of software modules regardless of their implementation language. Second, it provides a toolset extendable with new languages, which allows generation of the runtime verification modules for various implementation languages. These two features of E-Chaser reduce the developer's effort to utilize runtime verification in a multi-language setting. In addition, it improves the accuracy of verification because no software modules are excluded from verification due to unsupported programming languages; also, the individual properties of each subsystem as well as the synchronization properties of multiple subsystems can be specified and verified.

E-Chaser extends the Composition Filter Model [11] to support the verification of multiple-language software with the above-mentioned features. In the following, we explain the Composition Filter Model, and the E-Chaser extension to it; finally, we provide an overview of the E-Chaser architecture.

A. The Composition Filter Model

The Composition Filter Model aims at improving the compose-ability of object-oriented software. In such software, objects can send messages between each other, e.g. in the form of method calls or events. In the Composition Filter Model, these messages can be filtered, as shown in Fig.5. Each filter has a type, which implements the functionality that should be executed if the filter receives a message. For example, one may develop a filter type that verifies incoming messages to an object against a logical predicate.

Filters are grouped in so-called **filter modules**. A **superimposition selector** chooses a set of classes using a query language and applies (**superimposes**) a specified filter module to them. As a result, all messages sent to and received by all instances of those selected classes are subjected to the filters within the filter module.

The Composition Filter Model can be applied to any language that supports the notion of message passing between objects. In a non-object-oriented language such as C, the invocation of functions can be considered as messages that are passed between source files. This characteristic of the Composition Filter Model helps us make E-Chaser's specifications language-independent and the toolset extendable with new languages.

The idea of the Composition Filter Model was already implemented in our group in the Compose* tool [11], which provides a language- and platform-independent specification language and compiler for the Composition Filter Model.

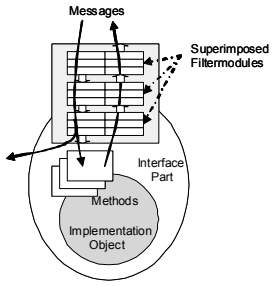

Figure 5. The Composition Filter Model

B. The E-Chaser Extention to the Composition Filter Model

E-Chaser aims at verifying those properties of software which can be expressed as a sequence of messages. At the first glance, this can be done via providing a filter type that verifies a sequence of messages expressed as, for example, regular expression, and executes an action if the sequence of messages is not satisfied at runtime.

With the current Composition Filter Model, it is only possible to specify and verify the message sequences of individual subsystems by defining individual filters for each of them. Therefore, the verification of synchronization properties among multiple subsystems remains impossible.

To support specification and verification of the synchronization properties, E-Chaser extends the Composition Filter Model with the notion of **synchronization messages** that are generated by filters, for example, to announce results of their functionality. E-Chaser also extends the Composition Filter Model with **synchronization filters**, which are superimposed on the individual filters defined for each subsystem. A synchronization filter specifies a synchronization property among individual filters in terms of their synchronization messages, and verifies the property at runtime.

C. The Overall Architecture of E-Chaser

In E-Chaser, both properties of individual subsystems and the synchronization properties are expressed as regular expression predicates over either normal messages or synchronization messages. In this manner, to implement E-

Chaser we extend the Compose* tool with a new filter type called *RuntimeVerificationFilterType* which verifies a regular expression predicate and executes an action if the predicate is not satisfied at runtime. E-Chaser also modifies the implementation of Compose* to support the synchronization messages and the synchronization filters. Fig.6 shows the overall architecture of E-Chaser.

In the specification layer, each specification is composed of two parts called *FilterModule*, and *Superimposition*. The part *FilterModule* contains a list of filters defined of the type *RuntimeVerificationFilterType*. To define each filter, one must specify (a) a regular expression predicate indicating the message-sequencing property that must be verified, (b) an action which must be executed when the verification of the regular expression predicate fails at runtime, and (c) a list of messages whose sequence must be verified.

For the specifications of the individual subsystems, the *Superimposition* specification defines a list of classes in each subsystem, whose message sequences must be verified. For the synchronization specifications, the *Superimposition* specifies a list of individual filters whose synchronization property must be verified. In both cases, the *Superimposition* specification applies the filter modules to the selected classes or filters.

Referring to Fig.1, filters define the specification in terms of messages, predicates and actions. The superimposition defines the software modules that generate the events.

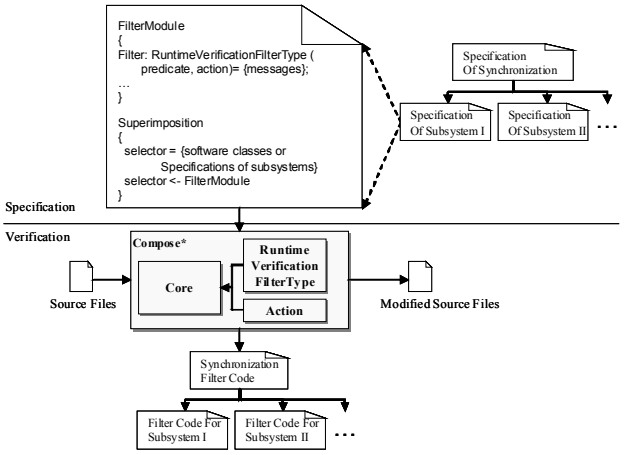

Figure 6. The overall architecture of E-Chaser

In the verification layer, E-Chaser extends the *Core* component of Compose* with the filter type *RuntimeVerificationFilterType*. The piece of functionality that is executed when the predicates fail at runtime is implemented in a module referred to as *Action*, which is provided for *Core*, as well.

The specifications and source files are input to Compose* which recursively generates code for the filters defined in the specifications, and superimposes them on the source files. Referring to the verification layer in Fig.1, the generated code for filters is equivalent to a combination of the monitors and the handlers; whereas the superimposition of filters is equivalent to the observers.

V. THE SPECIFICATION LAYER OF E-CHASER

Fig.7 shows the BNF grammar of the E-Chaser specification language. The details of this language are explained by examples in sub-sections *A* and *B*. In sub-section *A* the individual specifications of aTunes and MPlayer and in sub-section *B* the specification of synchronization are explained.

```
<SPECIFICATION>    ::= <FILTERMODULE> <SUPERIMPOSITION>
<FILTERMODULE>     ::= 'Filtermodule' <IDENTIFIER>'{'
                       <INTERNALS><INPUTFILTERS>'}'
<INTERNALS>        ::= 'Internals' (<IDENTIFIER> ':'
                       <IDENTIFIER> ';')+
<INPUTFILTERS>     ::= 'Inputfilters'<FILTER>+
<FILTER>           ::= <IDENTIFIER> ':
      RuntimeVerificationFilterType (Property=\' ' <REGEXP> '\',
      Action =\"<FQIDENTIFIER>'\')  ='<SELECTOR> ');'
<SELECTOR>         ::= '(Selector == /'\''<IDENTIFIER> '\''('as'
      <LITERAL>)? (',' '\''<IDENTIFIER>'\'' ('as' '\''<IDENTIFIER>'\''
      )?)* '/'('& Sender ==\" <FQIDENTIFIER>'\'')? ')'
      ( '|'<SELECTOR>)*
<SUPERIMPOSITION>::= 'Superimposition {' <SELECTORS>
<FILTERMODULES>'}'
<SELECTORS>        ::= 'Selectors' (<IDENTIFIER> '= {'
      <IDENTIFIER> '|' ('isModuleWithName' | 'isFilterWithName') '('
      <IDENTIFIER>',\"' <FQIDENTIFIER>'\'')'};')+
<FILTERMODULES>    ::= 'Filtermodules'
                       (<IDENTIFIER> '<-'<IDENTIFIER>';' )+
<REGEXP>           ::= <IDENTIFIER> | (<REGEXP> '*' | '+' | '?') |
                       (<REGEXP> (' ' | '|') <REGEXP>) |
                       '(' REGEXP ')'
<FQIDENTIFIER>     ::= <IDENTIFIER>('.' <IDENTIFIER>)*
<IDENTIFIER>       ::= 'A'.. 'Z' | 'a'.. 'z' | '_' ('A'.. 'Z' | 'a'.. 'z' | '_'
                       |'0'.. '9')*
```

Figure 7. A BNF grammar for E-Chaser specification language

A. The Specifications of aTunes and MPlayer

Fig.8 shows the specification of the message sequence for the subsystem aTunes in the language of E-Chaser. Lines 1 to 9 define a filter module called *aTunesSpecification*. In the part *Internals* of the filter module, a local variable called *logger*, represents an instance of the type *LogManager* that provides a general purpose logging functionality.

In the part *Inputfilters* (lines 4 to 8) a filter called *aTunesProperty* is defined of the type *RuntimeVerificationFilterType* which receives two input arguments called *Property* and *Action*. The argument *Property* is assigned with a regular expression predicate indicating that the messages *VolumeUp*, *UpdateGUI* and *WriteCommand* must follow each other for zero or more times, to increase the volume in aTunes correctly. The argument *Action* refers to the method *log*, accessible through the variable *logger*. In line 8 at the right-hand side of the assignment operator, the keyword *Selector* is used to specify the name of messages that must be filtered by *aTunesProperty*. Here, the messages *VolumeUp*, *UpdateGUI* and *WriteCommand*, which correspond to the execution of methods with the same name as the messages, are filtered. Lines 10 to 15 define the superimposition specification. In the part *Selectors*, we use the Prolog query language to select a set of modules (e.g. classes) in the software to which the filter modules must be applied. Line 12 selects all modules with the name *GUI* and assigns the possible results to *GUISelector*. In the part *Filtermodules*, we superimpose

aTunesSpecification and on the software modules selected by *GUISelector*. Likewise, Fig.9 defines the specification of the subsystem MPlayer.

```
1. Filtermodule aTunesSpecification{
2.  Internals
3.      logger:LogManager;
4.  Inputfilters
5.      aTunesProperty:RuntimeVerificationFilterType (
6.          Property='(VolumeUp UpdateGUI WriteCommand)*',
7.          Action='logger.log')
8.          = (Selector==['VolumeUp','UpdateGUI', 'WriteCommand']);
9. }
10. Superimposition{
11.  Selectors
12.      GUISelector = { C | isModuleWithName(C, 'GUI') };
13.  Filtermodules
14.      GUISelector <- aTunesSpecification;
15. }
```

Figure 8. Specification of aTunes in E-Chaser

```
1. Filtermodule MPlayerSpecification{
2.  Internals
3.      logger:LogManager;
4.  Inputfilters
5.      MPlayerProperty: RuntimeVerificationFilterType (
6.          Property ='(Read SetVolume)*', Action='logger.log')
7.          = (Selector==['Read', 'SetVolume']);
8. }
9. Superimposition{
10. Selectors
11.      CoreSelector = { C | isModuleWithName(C, 'Core') };
12.  Filtermodules
13.      CoreSelector <- MPlayerSpecification;}
```

Figure 9. Specification of MPlayer in E-Chaser

B. The Specification of Synchronization

In E-Chaser, *RuntimeVerificationFilterType* generates three synchronization messages called **Started**, **Succeeded** and **Failed,** which respectively indicate the start and the results of evaluation of a regular expression predicate. The synchronization property of multiple message-sequences must be specified using these messages. For example, one can define the synchronization of aTunes and MPlayer as follows:

After the specified sequence of messages for aTunes occurs successfully, the specified sequence of messages for MPlayer must start.

Fig.10 shows the specification of the above synchronization property. Similar to the previous specifications, lines 1 to 3 define a filter module and its local variable, and lines 4 to 11 define the filter *SynchProperty* of the type *RuntimeVerificationFilterType*. In line 6, a regular expression predicate is defined which implies the message *aTunesSucceeded* must be followed by the message *MPlayerStarted*, in zero or more times. In line 7, the argument *Action* is being assigned. Lines 8 and 9 select the message *Succeeded* that is sent by the filter *aTunesProperty*, and names the message to *aTunesSucceeded*. Lines 10 and 11 select the message *Started* that is sent by the filter *MPlayerProperty*, and names the message to *MPlayerStarted*.

In the *Superimposition* section, *aTunesSpecSelector* and *MPlayerSpecSelector* make use of our Prolog predicate *isFilterWithName* to select the filters *aTunesProperty* and *MPlayerProperty* that are defined in the specifications *aTunesSpecification* and *MPlayerSpecification*, respectively; and the superimposition is done in lines 20 and 21. In line 20, by enclosing *aTunesSpecSelector* and *MPlayerSpecSelector* in the set notation, *SynchronizationSpecification* is superimposed on them.

```
1. Filtermodule SynchronizationSpecification{
2.  Internals
3.      logger:LogManager;
4.  Inputfilters
5.      SynchProperty: RuntimeVerificationFilterType (
6.        Property= ' (aTunesSucceeded MPlayerStarted)* ',
7.        Action='logger.log') =
8.        (Selector == ['Succeeded' as 'aTunesSucceeded'] &
9.        Sender=='aTunesSpecification.aTunesProperty')
10.       | Selector ==['Started' as 'MPlayerStarted'] &
11.       Sender == 'MPlayerSpecification.MPlayerProperty') ;
12. }
13. Superimposition{
14.  Selectors
15.      aTunesSpecSelector={C|isFilterWithName(C,
16.                          'aTunesSpecification.aTunesProperty') };
17.      MPlayerSpecSelector = { C | isFilterWithName(C,
18.                          'MPlayerSpecification.MPlayerProperty') };
19.  Filtermodules
20.      {aTunesSpecSelector, MPlayerSpecSelector}<-
21.                          SynchronizationSpecification;
22.}
```

Figure 10. Specification of synchronization

VI. The Verification Layer of E-Chaser

In this section, we explain the verification layer of E-Chaser from two points of views: compile-time and runtime. Along with the compile-time view, the architecture of Compose* is explained; in the sub-section on the runtime view, the communication between filters is discussed.

A. The Compile-Time View

Fig.11 provides a global overview of the Compose* modules and their interaction to generate code for filters from the specifications. The modules presented in this figure are explained from left to right in the following.

RuntimeVerificationFilterType (and in general each filter type in Compose*) is composed of language-specific code generators and a language-independent meta-information. The code generators of a filter type produce code for filters of this type, and the meta-information provides a description of the filter type, e.g. its name.

The language-dependent module *Action* provides the functionality that must be executed when the evaluation of the predicate fails at runtime. *RuntimeVerificationFilterType* along with the *Specifications*, *Action* and the *SourceFiles* of the subsystems are input to the core modules of Compose*. The modules in the core part of Compose* are categorized to language-dependent and language-independent modules, of which the language-dependent ones must be implemented for any supported implementation language.

In the core part of Compose*, a shared repository is used by the various compiler modules. The filter type

RuntimeVerificationFilterType is input to the module *FilterTypeParser* that stores the meta-information along with references to the code generators in *Repository*.

Specifications and *Action* are input to the module *SpecificationParser* that checks the syntactical correctness of specifications, translates the specifications to a set of Prolog facts to be used later on for superimposing synchronization filters, resolves the references to *Actions* from within the specifications, and stores the Prolog facts and the references to *Actions* in *Repository*. In addition to the Prolog facts, *SpecificationParser* stores the Prolog queries of the selectors (see for example line 11 in Fig.9) in *Repository*.

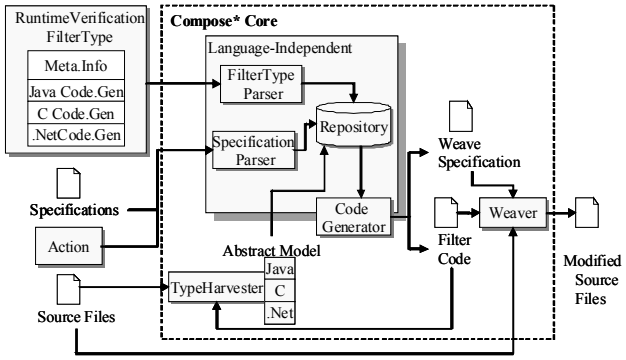

Figure 11. Overall architecture of Compose*

To support various implementation languages, Compose* defines an abstract model of the languages to which both object-oriented and procedural languages can be mapped. The module *TypeHarvester* analyzes *SourceFiles*, produces a representation of them in the abstract model and stores it as Prolog facts in *Repository*.

The module *CodeGenerator* translates the filters to executable code and superimposes them on *SourceFiles*. There are multiple ways to translate the filters to executable code, ranging from completely static to completely dynamic [11]. With the static execution, the filter code is generated in the same language as *SourceFiles* and is merged with *SourceFiles*. In this case, the composition filters become an integral part of *SourceFiles*. The dynamic execution allows the filters to be modified at runtime. To support dynamic execution, a **runtime interpreter** is needed to evaluate the filter specification. This runtime interpreter needs to be called from the locations in *SourceFiles* on which the filters are superimposed on. The static introduction of the composition filters offers less flexibility and evolvability than their dynamic evaluation, but it does offer better runtime performance. In the current implementation of Compose*, the dynamic approach is supported for the Java and .Net languages and the static one for C.

The module *CodeGenerator* is composed of a language-independent and a language-dependent part. To generate code for the ordinary, i.e. non-synchronizing, filters, the language-independent part evaluates the Prolog queries of the selectors against the Prolog facts stored in *Repository* and creates a *WeaveSpecification*. The latter specifies the places in the abstract model of *SourceFiles* on which the filter code must be superimposed. The language-dependent part invokes the code generator of the filter type. For the Java language, *JavaCodeGen* produces calls to the runtime interpreter. For the C language, *CCodeGen* produces code that implements the whole functionality of the filters. *WeaveSpecification* and the generated *FiltersCode* are input to the language-dependent module *Weaver* that inserts *FiltersCode* at the appropriate places in *SourceFiles*.

E-Chaser makes use of the Java-RMI technology [18] to support synchronization filters. In this manner, it extends the module *CodeGenerator* to produce code for the synchronization filters in a way that they are executed as RMI servers in separate processes, and to produce code for the ordinary filters to communicate with the synchronization filters as RMI clients. To superimpose the synchronization filters on the ordinary filters, the generated code for the ordinary filters are given to the module *TypeHarvester*; and *CodeGenerator* does the superimposition in the same way as for the ordinary filters.

As we discussed earlier, the generated code for the ordinary filters may not always be in Java. In this case, *CodeGenerator* produces intermediate software modules that enable the non-java code to communicate with the synchronization filters through Java-RMI.

B. The Runtime View

The runtime view of E-Chaser for our running example is shown in Fig.12. Our running example consists of two processes called aTunes and MPlayer, and because we have defined a synchronization filter (see Fig.10) there will be another process called *Synchronization*, which executes the filter *SynchProperty* and its enclosing filter module as an RMI server. In the rest of this discussion, for the sake of brevity, we do not distinguish between the name of classes and their instances.

In the process *aTunes* there is an instance of the class *GUI*, whose runtime behavior is managed by *JavaRuntimeInterpreter*. Upon instantiation of *GUI,* an instance of the filter module *aTunesSpecification* is created by *JavaRuntimeInterpreter* and is superimposed on *GUI*. The instantiated filter module contains the local variable *logger* besides the filter *aTunesProperty*. *JavaRuntimeInterpreter* also superimposes a RMI proxy of *SynchProperty* on *aTunesProperty*.

The process *MPlayer* executes the C module *Core*, into which the filter module *MPlayerSpecification* is merged. Since Compose* statically introduces filters for modules written in C, there is no runtime interpreter for the C software. To facilitate sending synchronization messages from C software to the synchronization filter through Java-RMI, E-Chaser makes use of an intermediate Java object called *Communicator*, on which a proxy of the filter *SynchProperty* is superimposed. The C module *Core* makes use of Java Native Interface (JNI) [18] to send the synchronization messages to *Communicator*, which accordingly transfers them to *SynchProperty* via Java-RMI.

Here, we use two scenarios to discuss the runtime view of our running example. First, we assume that all the specified

messages occur in the specified order. In E-Chaser, the first message that satisfies the specified sequence of message is considered as the **start-point** message upon which verification of the sequence started. Here, the invocation of *VolumeUp* on *GUI* is considered as the start-point message for the verification of the subsystem aTunes. Before the execution of *VolumeUP*, *JavaRuntimeInterpreter* sends the message *VolumeUP* to the filter *aTunesProperty*, which verifies the messages against the specified regular expression. Since the message does not violate the regular expression, *JavaRuntimeInterpreter* executes the method *VolumeUp* on *GUI*. After all the specified messages occur in the specified order, *aTunesProperty* generates the synchronization message *Succeeded*. The message is received by the proxy of *SynchProperty*, which accordingly transfers it to the remote filter *SynchProperty* via Java-RMI. *SynchProperty* verifies the message and waits for arrival of the synchronization message *Started* from the process *MPlayer*.

The process *MPlayer* receives the command *VolumeUp* from aTunes; therefore, the function *Read* in *Core* is about to be executed. Before the execution of *Read*, the verification of the regular expression specified by the filter *MPlayerProperty* is started. That results in generating the synchronization message *Started* by *MPlayerProperty*. The message *Started* is sent via Java-JNI to *Communicator*, which accordingly transfers it to the remote filter *SynchProperty* through the proxy of *SynchProperty*. By receiving the message *Started*, the verification of the regular expression in *SynchProperty* also succeeds, which means the synchronization property of aTunes and MPlayer is satisfied.

Likewise, before execution of the function *SetVolume* in *Core*, *MPlayerProperty* verifies it against the regular expression.

As our second scenario, let us assume that the message sequence of aTunes occurs as expected, but due to some communication problems, the command *VolumeUp* is not received by MPlayer. This is a typical example in which some of the expected messages are missing and at some point, we have to conclude that they will never occur and report their absence as a violation of the property. The point at which this conclusion must be done is specified by a so-called **end-point** message, which is generated by the software before exaction of the software terminates. In this example, if *SynchProperty* does not receive the message *Started* before the execution of software terminates, it reports that the synchronization property is violated and it executes the method *log* on *logger* to dump some information about the violated property.

VII. Operational Semantics

In the specification layer of E-Chaser, the specification S for multiple-language software is a collection of specifications S_p of individual properties and specifications S_s of synchronization properties:

$$S := \{S_{p0}, S_{p1}, \ldots, S_{s0}, S_{s1}, \ldots\} \quad (1)$$

Each S_{pi} contains a specification of the individual property P_i for a software module or a subsystem. The property P_i is specified as a regular expression with the grammar defined in (2) [19].

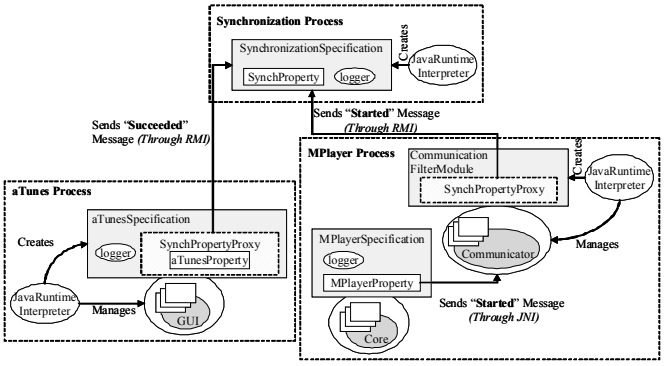

Figure 12. Runtime view of E-Chaser for the running example

$$
\begin{aligned}
&P_i := REGEXP \\
&REGEXP := identifier \quad // Message\ name \\
&REGEXP := REGEXP\ REGEXP \quad // Concatenation \\
&REGEXP := REGEXP\text{'}|\text{'}REGEXP \quad //Boolean\ Or \\
&REGEXP := REGEXP\text{'}*\text{'} \quad //Repetition\ 0\ or\ more\ times \\
&REGEXP := REGEXP\text{'}+\text{'} \quad //Repetition\ 1\ or\ more\ times \\
&REGEXP := REGEXP\text{'}?\text{'} \quad //\ 0\ or\ 1\ time \\
&REGEXP := \text{'}(\text{'}REGEXP\text{'})\text{'} \quad //Grouping
\end{aligned}
\quad (2)
$$

In the verification layer, the rules in [19] are used to translate P_i to the deterministic finite state automaton A_i, which is defined as in (3).

$$A_i := \{Q, \Sigma, d, q_0, F, Q\} \quad (3)$$

Where:

- Q is a finite set of states
- Σ is a finite set of messages, specified by P_i, which must be filtered for verification.
- q_0, a member of Q, is the start state.
- F, a subset of Q, is the set of final (or accepting) states.
- d is a transition function that takes a filtered *message* $\in \Sigma$ and a *state* $\in Q$ as its input, and returns a *state* $\in Q$.
- $\Theta \in Q$ is the trap state, which is reached by all messages that do not satisfy P_i.

The verification is defined as an algorithm in Fig.13, which for every property P_i and the failure-detection/tolerance functionality $Action_i$, verifies the message m with the generated automaton A_i, starting from the initial state q_0. In lines 2 to 23, the verification is performed until the **end-point** message, which indicates execution of the software is about to be finished, is received. In line 4, the transition function d is invoked to verify m in the current state of the automaton. In lines 5 and 6, if the current state is q_0 and m leads to the trap state Q, m is just ignored by not changing the current state of the automaton. This implies that only the first accepted message in the sequence of messages is considered as the **start-point** message, and all other messages before the start-

point message are ignored. Upon this first match, in lines 7 to 10, the message m_{Start} is generated and *currentState* is updated with *targetState*.

```
1. Algorithm Verification_Pi,Actioni (m, Ai)
2.  While (m <> end-point)
3.  {
4.     targetState = d (m, currentState)
5.    If ( currentState = q0 ) and (targetState = Q)
6.            Ignore m
7.    Else if ( currentState = q0 ) and (targetState ≠ Q) {
8.            Generate mStart for Pi
9.            currentState := targetState
10.      }
11.   Else if ( currentState ≠ q0 ) and (targetState = Q) {
12.           Generate mFailed for Pi
13.           Execute Actioni
14.           If (Pi is enclosed by '*' or '+')
15.             currentState := q0
16.   }
17.   Else if ( currentState ≠ q0 ) and (targetState ⊆ F){
18.            Generate mSucceeded for Pi
19.            currentState := targetState
20.    }
21.    Else
22.           currentState := targetState
23. }
24. If (currentState ⊄ F) {
25.     Generate mFailed for Pi
26.      Execute Actioni
27. }
28. End
```

Figure 13. E-Chaser verification algorithm

If the automaton is not in the initial state and the target state is the trap state Q (lines 11 to 16), the synchronization message m_{Failed} is generated, $Action_i$ is executed. In addition, if the sequence of messages specified for P_i is enclosed in repetitive operators (e.g. $P_i := (abc)^*$), the verification restarts from the beginning by assigning q_0 to *currentState*. This implies that the specified sequence of messages for P_i may occur again during runtime and therefore must be verified; although once it has been violated. In lines 17 to 20, if m led to a final state, the synchronization message $m_{Succeeded}$ is generated and *currentState* is updated with the *targetState*. In lines 21 and 22, if none of the above cases occurs, *currentState* is just updated with *targetState*; this means that the verification is still in intermediate states and we cannot reason about succeed or failure of P_i.

If the end-point message arrives and the automaton is not in a final state (lines 24 to 28), we conclude that some messages were missing; therefore, the synchronization message m_{Failed} is generated and $Action_i$ is executed.

Each specification S_{si} defines the synchronization properties P_s in the same way as for P_i, except that the regular expression is defined over the generated synchronization messages during the verification of the individual properties P_i.

The synchronization messages are only generated when the verifications of any P_i starts, succeeds or fails. Assume that there are two properties $P_i := m_0 m_1 m_2$ and $P_j := m_3 m_4 m_5$ and P_j must be synchronized with P_i on m_1. According to Fig.13, m_1 is verified in lines 21 and 22; therefore, no synchronization message is generated upon which P_j is synchronized with P_i. To support such synchronization properties, one must decompose P_i into smaller regular expression predicates, say P_i', P_i'', and so on, in a way that upon verification of m_1, either m_{Start} or $m_{Succeeded}$ is generated. Then, the synchronization property must be defined for P_i', P_i'' and P_j. Decomposition of a regular expression into smaller ones can be done using the regular expression operations explained in [19].

In the verification layer, P_s is evaluated using a similar algorithm as for P_i, except input messages to the algorithm are the synchronization messages generated during verification of any Pi, and no synchronization message is generated during the verification of P_s.

VIII. DISCUSSION

Complex software, especially embedded software, is composed of multiple collaborating subsystems that provide different functionalities of the software. According to the functionality of each subsystem, different languages may be employed to facilitate implementation of the subsystem. In our research group, we deal with various examples of such software. Philips MRI software, ASML wafer scanner and Océ printer software are three of them [20].

Despite the increasing number of such software, we realized that in the runtime verification literature there is no single runtime verification system that can be applied to such software. Therefore, we developed E-Chaser and utilized its first working prototype to verify some properties of our case studies. The working prototype of E-Chaser along with some examples can be found in [21]. Below, we discuss the quality attributes *expressiveness*, *effort reduction* and *failure-handling capability*, and the extent to which E-Chaser can meet them.

A. Expressiveness

Due to the popularity of the regular expression formalism in the existing runtime verification systems, we also chose this formalism to express the properties of the software. During our evaluation with different case studies, we realized that despite the simplicity of the regular expressions, they are not sufficient to express the complex properties that are best expressed using other formalisms. For example, context-free grammars are one of the best-suited formalism to express structured properties that refer to the call stack of the program [9, 22]. Therefore, we believe that E-Chaser must be extended to support multiple formalisms. Nevertheless, the extension only influences some part of our specification language (i.e. definition of properties in filters) and the rest of the specifications can easily be reused for new formalism. In the implementation of E-Chaser, the logic to verify properties defined in a specific formalism is modularized in the filter type *RuntimeVerificationFilterType*; therefore, to support new formalisms we only need to change this filter type.

Every message, which is sent from or received by an object, may also change values of variables in the object. Hence, in order to verify that software provides correct service, both the sequence of messages and the changes to the values must be verified. To verify the values, they must, similar to the message, be filtered from the software and appropriate assertions must be specified to verify correctness of them. In

the E-Chaser implementation, for every filtered message, the values of its parameters are available to be verified. However, currently the specification language of E-Chaser is not expressive enough to define value assertions.

B. *Effort Reduction*

With E-Chase, we reduce the developer effort of verifying multiple-language software by three means. First, it supports language-independent specifications; hence, a developer must only learn one specification language regardless of the implementation language of the software. Second, it supports automatic generation of the runtime verification modules (i.e. filter code) from the specifications, and automatic modification of the software to work with the runtime verification modules. Third, the fact that in E-Chaser individual specifications must be defined for individual subsystems increases the reusability of the specifications; hence, it reduces a developer's effort to specify properties of the software. For example in our running case study, if aTunes is replaced with new user-interface software, the specification of aTunes and the synchronization must be replaced with new specifications whereas the specification of MPlayer can be reused.

C. *Failure-Handling Capability*

E-Chaser, in the first place, aims at detection of failures at runtime. However, it also provides a rather lightweight **global failure handling** mechanism by associating one failure-detection/tolerance action to each specified property. The failure-handling mechanism of E-Chaser can be improved by supporting a **local failure handling** mechanism in which dedicated actions are executed for each detected failure. For example, for every missing message in a sequence of messages, one may provide a specific action that generates the missing message and sends the message when it is expected. In our group, we perform dedicated research on runtime recovery techniques [23] which we aim to combine with the failure-detection capability of E-Chaser.

IX. Conclusion and Future Work

In this paper, we discussed our observed problems in the runtime verification of multiple-language software, and we proposed E-Chaser to address these problems. The idea of Composition Filters, upon which E-Chaser is based, made it possible to support multiple-language environments, and to provide language-independent specifications and a toolset extendable with new languages. E-Chaser provides a filter type for runtime verification of sequences of messages, and extends the Composition Filter Model with the notion of synchronization messages and synchronization filters to support verification of the synchronization properties of multiple subsystems. A working prototype of E-Chaser has been developed and has been used to verify properties of some case studies.

In future work we plan to evaluate E-Chaser performance and runtime overhead for case studies that are more complex. Furthermore, we will improve the expressiveness of E-Chaser specifications by supporting the language-independent value assertions. We will also investigate on replacing the regular expression formalism with more powerful ones, in addition to supporting local failure-handling mechanisms.

Acknowledgement

We acknowledge the feedback from the discussions with our TRADER project partners from Embedded Systems Institute.

References

[1] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," IEEE transaction on dependable and secure computing, vol.1, 2004, pp. 11- 33.

[2] N. Delgado, A. Quiroz Gates, and S. Roach,"A taxonomy and catalog of runtime software-fault monitoring tools," IEEE transactions on software engineering, vol.30. 2004.

[3] M. Kim, M.Viswanathan, S. Kannan, I. Lee, and O. Sokolsky, "Java-MaC: a run-time assurance approach for Java programs," Formal methods in system design, vol. 24. Springer-Verlag, 2004.

[4] X. Logean, F. Dietrich, H. Karamyan, and S. Koppenhofer, "Run-time monitoring of distributed applications," Proceedings of Middleware. Springer-Verlag, 1998.

[5] CORBA, http://www.corba.org

[6] F. Chen, and G. Rosu, "MOP: an efficient and generic runtime verification framework," OOPSLA, Montreal, Quebec, Canada, 2007.

[7] K. Havelund, "Runtime verification of C programs," TestCom/FATES., LNCS, vol. 5047, 2008.

[8] C. Allan, P. Avgustinov, A.S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "Adding trace matching with free variables to AspectJ," OOPSLA, Oregon, USA, 2005.

[9] M.M. Benjamin, L. Monica, and S. Lam, "Finding application errors and security flaws Using PQL: a program query language," OOPSLA, Oregon, USA, 2005.

[10] D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim, "Jass – java with assertions," RV, Paris, France, 2001.

[11] A.J. de Roo, M.F.H. Hendriks, W.K. Havinga, P.E.A. Durr, and L.M.J. Bergmans,"Compose*: a language- and platform-independent aspect compiler for composition filters," in WASDeTT, 2008.

[12] aTunes, http://www.atunes.org

[13] MPLayer, http://www.mplayerhq.hu

[14] K. Havelund, and G. Rosu, "An overview of the runtime verification tool Java PathExplorer," in Formal methods in system design, vol. 24, 2004, pp. 189-215.

[15] H. Barringer, A. Goldberg, K. Havelund and K. Sen, "Rule-based runtime verification," LNCS, vol. 2937, 2004.

[16] R. Laddad, AspectJ in Action: Practical Aspect-Oriented Programming, Manning Publications, 2003.

[17] D. Harel, "Statecharts : a visual formalism for complex systems," science of computer programming, vol. 8, 1987, pp. 231-274.

[18] Java, http://java.sun.com

[19] J. E. Hopcroft, R. Motwani, and J.D. Ullman, Introduction to Automata Theory, Languages, and Computation. Addison Wesley, 2nd edition, 2000.

[20] ESI, http://www.esi.nl

[21] E-Chaser, http://trese.cs.utwente.nl/e-chaser/

[22] P.O. Meredith, D. Jin, F. Chen, and G. Rosu, P.O. "Efficient Monitoring of Parametric Context Free Patterns", ASE'08, L'Aquila, Italy, 2008.

[23] H. Sözer, Architecting Fault-Tolerant Software Systems. PhD thesis, University of Twente, 2009.