Correct-by-construction implementation of runtime monitors using stepwise refinement [⋆]

Teng Zhang[1], John Wiegley[2], Theophilos Giannakopoulos[2], Gregory Eakman[2]
Clément Pit-Claudel[3], Insup Lee[1], and Oleg Sokolsky[1]

[1] University of Pennsylvania, Philadelphia PA 19104, USA,
{tengz,lee,sokolsky}@cis.upenn.edu
[2] BAE Systems, Burlington MA 01803, USA,
{john.wiegley, theo.giannakopoulos, gregory.eakman}@baesystems.com
[3] MIT CSAIL, Cambridge MA 02139, USA,
cpitcla@csail.mit.edu

Abstract. Runtime verification (RV) is a lightweight technique for verifying traces of computer systems. One challenge in applying RV is to guarantee that the implementation of a runtime monitor correctly detects and signals unexpected events. In this paper, we present a method for deriving correct-by-construction implementations of runtime monitors from high-level specifications using Fiat, a Coq library for stepwise refinement. SMEDL (Scenario-based Meta-Event Definition Language), a domain specific language for event-driven RV, is chosen as the specification language. We propose an operational semantics for SMEDL suitable to be used in Fiat to describe the behavior of a monitor in a relational way. Then, by utilizing Fiat's refinement calculus, we transform a declarative monitor specification into an executable runtime monitor with a proof that the behavior of the implementation is strictly a subset of that provided by the specification. Moreover, we define a predicate on the syntax structure of a monitor definition to ensure termination and determinism. Most of the proof work required to generate monitor code has been automated.

Keywords: Runtime Monitor, SMEDL, Formal semantics, Coq, Stepwise refinement

1 Introduction

Runtime verification (RV) [1] is a lightweight technique for correctness monitoring of critical systems. The objective of RV is to check if a run of the system (referred as a target system in the remainder of the paper), usually abstracted as a trace of events either from the execution or the logging information, satisfies or violates certain properties. Properties to be checked using RV are usually

[⋆] This work is supported in part by the Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under contract FA8750-16-C-0007 and by ONR SBIR contract N00014-15-C-0126.

specified by high level languages, such as temporal logics or state machines. Specifications are then converted into executable monitor code by either a code generator or manual effort. However, informal code generation processes are usually error-prone, and the generated monitor code may not adhere to its specification. During execution, an incorrect monitor may not detect property violations, which can lead to serious consequences in safety critical systems. As a result, it is desirable to use a formal procedure to achieve correct-by-construction implementation of monitors.

This paper presents a method for generating correct-by-construction implementations of runtime monitors written in SMEDL [2], a state-machine-based DSL (domain specific language) for RV. Our method is based on Fiat [3], a powerful deductive synthesis framework embedded in the Coq proof assistant [4]. The core idea in Fiat is to separate declarative specifications from concrete implementations. Users start by embedding their DSL into Coq, so that each DSL program is understood as a mathematical description of the set of results it may return. Using stepwise refinement, each program can be translated into a correct-by-construction executable implementation.

High-level specifications of monitors are attractive because they succinctly describe what monitors should do: implementation details are crucial for performance, but they can be determined separately, along with a proof of preservation of semantics between the specification and implementation. One challenge is to design semantic rules that can be used smoothly in the specification, while remaining amenable to refinement so that such preservation can be proved without excess difficulty.

Additionally, to generate a correct monitor, we have to ensure that the monitor specification is well-formed. In this paper, we require that each monitor satisfies two properties: *termination* and *determinism*. Termination is important because if a monitor goes into infinite loop, or gets stuck during the execution, it will not be able to receive events from the system and catch property violations. Determinism ensures that the monitor always produces the same output given the same input and current state. The code generation process thus needs to be able to detect and reject any "bad" monitor specification that may get stuck during execution.

To overcome the challenges above, we provide a solution for constructing runtime monitor implementation using the Fiat framework. The contributions of this paper are the following:

- We present an operational semantics for SMEDL monitors written in a relational way that ensures that the functionality of the specification and the implementation are separated. This lays a foundation for generating correct code using Fiat.
- We define a predicate on the definition of SMEDL monitors to ensure termination and determinism. Only a well-formed monitor can be extracted into executable code through the Fiat framework.
- We implement a complete procedure from the design of specifications to the generation of correct-by-construction runtime monitors using Fiat, illus-

trating how deductive synthesis can be used to formally derive trustworthy monitors.

The code generation process presented in this paper is shown in Fig 1. First, we define a general declarative specification to describe the behavior of a monitor reacting to input events. This specification is independent of specific monitor definitions. It is then refined into a general function using Fiat. To derive executable code for a specific monitor, the user needs to specialize this general function using a definition of that monitor along with a proof of its well-formedness. A Haskell program is extracted and rendered into a monitor to check properties of the target system. Since most of the proof work is automated by auxiliary decision procedures and tactics, applying this methodology does not lead to a heavy workload. Furthermore, efficient monitor code can be generated simply by picking other implementations, which are guaranteed by Fiat to adhere to the behavior described by the specification.

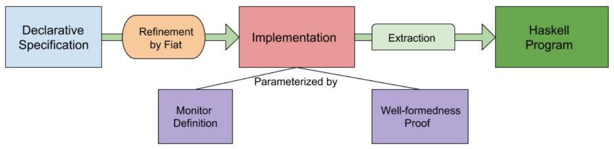

Fig. 1. Code generation process

The paper is organized as follows. Section 2 introduces basic concepts of the Fiat framework and SMEDL. Section 3 and 4, respectively present the operational semantics of SMEDL and the monitor definition predicates that ensure termination and determinism. Section 5 and 6 present the process of refinement using Fiat and illustrate the usability of this method by a case study. Section 7 summarizes related work. Section 8 concludes the paper and presents the future work. The code for this paper can be downloaded from the internet[1].

2 Preliminaries

Overview of Fiat. Stepwise refinement derives executable programs from nondeterministic specifications. In each step, some details of the computation are decided upon, proceeding this way until a computable program is derived. Each refinement step must not introduce new behavior: the values that a refined program may produce must be a subset of the values allowed by the specification. Fiat is a stepwise refinement framework, providing a semi-automatic way of deriving correct and efficient programs. Here "semi" means that while the

[1] `https://gitlab.precise.seas.upenn.edu/tengz/smedl-fiat-code`

derivation process is automatic, it depends on manually verified refinement lemmas, specific to the domain that Fiat is applied to. This section briefly gives an overview of Fiat. Readers can refer to [3,5,6] for more information.

Important syntax structures in Fiat. In Fiat, specifications are logical predicates characterizing allowable output values. These specifications are called computations, and written in the non-determinism monad: deterministic programs can be lifted into computations using the "ret" combinator, computations can be sequenced using the "bind" combinator (written "$x \leftarrow c_1; c_2(x)$"), and a nondeterministic choice operator written $\{a|P\ a\}$ is used to describe programs that may return any value satisfying a logical predicate P. Concretely, the result of binding two computations c_1 and c_2 as shown above is simply the set $\{y|\exists x, x \in c_1 \wedge y \in c_2(x)\}$.

Fiat computations are organized into an Abstract Data Type (ADT), a structure used to encapsulate a set of operations on an internal data type. In Fiat, an ADT contains an internal representation type (denoted as *rep*), a list of constructors for creating values of type *rep*, and a list of methods operating on values of type *rep*. A well-typed ADT guarantees that *rep* is opaque to client programs using the operations of the ADT.

Refinement calculus in Fiat. Refinement in Fiat is the process of transforming an ADT into a more deterministic ADT, involving refining all constructors and methods defined in ADT and picking an efficient internal representation using data refinement [7] of *rep*. When refining an expression, a partial relation $c_1 \supseteq c_2$ must be preserved for each refinement step, meaning that the possible values of expression c_2 must be a subset of the possible values of expression c_1. For the data refinement part, changes of internal representation are justified using a user-selected abstraction relation, so that if the internal states of two ADTs are related, calling their methods must preserve the relation and produce the same client-visible outputs. Adding the abstraction relation r to the partial relation $\supseteq$ of refinement on expression, Fiat uses $\supseteq_r$ to represent the relation to be preserved for each refinement step: $I_1 \supseteq_r I_2 \supseteq_r \dots \supseteq_r I_i$ where I_1 is the initial ADT and I_i is a *fully refined* (i.e. deterministic) ADT.

SMEDL concepts. A SMEDL monitor is a collection of *scenarios*. Each scenario is an Extended Finite State Machine (EFSM) [8] in which the transitions are performed by reacting to events. By specifying a set of scenarios in a SMEDL monitor, the monitor can check the behavior of a certain aspect of the system from a variant levels of abstraction with a clearer separation. Scenarios interact with each other using shared state variables or by triggering execution of other scenarios through raised events. There are three types of events: *imported*, *exported* and *internal*. Imported events, which are responsible for triggering the execution of a monitor, are raised from the target system or by other monitors; exported events are raised within the monitor and are then sent to other monitors; internal events are used to trigger transitions, but are only seen and processed within a given monitor. Each transition is labeled with a triggering event and attached to a guard condition and a list of actions to be executed after the transition. Primitive data types, such as arithmetic and logical operations, are supported in SMEDL. The abstract syntax of SMEDL is given below.

A monitor is a 3-tuple $\langle V, \Sigma, S\rangle$, where V is a set of state variables, Σ is a set of event declarations and S is a non-empty set of scenarios for the monitor. The event declaration is a three-tuple $\langle \mathit{eventType}, \mathit{eventName}, \mathit{attributeTypes}\rangle$ where *eventType* is an enumeration over three types of event introduced above; *attributeTypes* represents the list of types of attributes of the event.

A scenario is a 5-tuple $\langle n, Q, q_0, E, \delta\rangle$ where n, Q, and q_0 are respectively the identity, the set of states and the initial state, E is the set of *alphabets*—events that can trigger the transitions of the machine and δ is the set of transitions of the scenario.

A transition is a four-tuple $\langle \mathit{qsrc}, \mathit{qdst}, \mathit{stpEv}, \mathcal{A}\rangle$ where *qsrc* and *qdst* are the source and target state of the transition, *stpEv* denotes the triggering event of the transition, typed with *eventInstance* which is a three-tuple $\langle \mathit{event}, \mathit{eventArgs}, \mathit{ev}$ $\mathit{entGuard}\rangle$. *event* is a reference to the corresponding event declaration, *eventArgs* is defined as the list of local variable names, and *eventGuard* is an expression guarding the transition. Both the state variables and local variables can be used in the *eventGuard*. The set $\mathcal{A}$ consists of statements to be executed immediately after the transition, which can either update state variables or raise events. Note that we assume all transitions in the scenario are complete.

3 An Operational Semantics of SMEDL

This section proposes the formal semantic rules for a single monitor. When an imported event is sent to a monitor, state transitions within the monitor are triggered. Actions attached to transitions can raise internal or exported events. Internal events are used to trigger further transitions in other scenarios. After all triggered transitions are completed, exported events are output and the monitor waits for the next imported event. This process is denoted as a *macro-step*, which cannot be interrupted by other imported events. Each scenario can execute its transition at most once in a single macro-step so that there is no infinite loop of interaction between scenarios. We introduce a data structure *configuration* to describe the dynamic state of a monitor.

Configurations. A configuration is a five-tuple $\langle \mathit{MS}, \mathit{DS}, \mathit{PD}, \mathit{EX}, \mathit{SC}\rangle$. *MS* denotes the mapping from scenarios to their current states; *DS* is a well-typed mapping from state variables to values; *PD* is the set of pending events to trigger transitions within the monitor; and *EX* is the set of raised, exported events. Elements in both sets are events binding with actual attribute values, denoted as *raisedEvents*. *SC* is the set of scenarios executed during the the current macro-step and its corresponding triggering events. Each configuration *conf* relates to a monitor *M*, denoted as conf_M. The subscript is omitted in the remainder of the paper whenever the context is clear.

A macro-step is constructed by chaining a series of consecutive *micro-steps*. Each micro-step is the synchronous composition of a set of transitions on scenarios with the same triggering event, constructed by the interleaved application of the *basic rule* and the *synchrony rule*. The chaining of micro-steps is performed by applying the *chain merge rule*.

Basic rule. The basic rule is applied to a state machine whenever a transition is triggered by a pending event. In the definition below, the scenario performing the transition is *mh*, *conf* denotes the configuration before applying the rule, and $conf'_{mh}$ denotes the configuration after applying the rule, on *mh*.

$$\frac{\begin{array}{c} tr : s1 \xrightarrow{e\{a\}}_{c} s2 \\ valid(tr, conf, mh) \\ conf'_{mh} = updateConfig(mh, conf, tr) \end{array}}{conf \xrightarrow{e} conf'_{mh}}$$

tr is the enabled transition from *s1* to *s2* by *e*; *a* is the set of actions for *tr*; and *c* is the guard. *valid* tests the validity of *tr* under the configuration *conf*, which includes: 1) *tr* is the transition of *mh*, 2) current state of *mh* is *s1* and 3) *c* evaluates to true for current *DS* and attribute values of the event, 4) *e* is in *PD*, and 5) *mh* is not in *SC*. When *tr* is taken, *conf* is updated by executing the function *updateConfig*, denoted as $conf \xrightarrow{e} conf'_{mh}$. The update includes: 1) *mh* transitions to *s2* and is put into *SC*, 2) *DS* is updated by the actions in the transition, 3) *e* is removed from *PD*, and 4) raised events are respectively added to *PD* and *EX* according their types.

Synchrony rule. One or more scenarios are enabled by a triggering event from a source configuration. The basic rule creates new configurations for each scenario by taking these transitions. The synchrony rule then combines scenario's resulting configuration into a new configuration. Combination of two configurations *conf1* and *conf2* under the origin configuration *conf* is defined below.

– $\forall mh \in S$,

$$MS_{conf'}(mh) = \begin{cases} MS_{conf_1}(mh) & , MS_{conf_1}(mh) = MS_{conf_2}(mh) \\ MS_{conf_1}(mh) & , MS_{conf_1}(mh) \neq MS_{conf}(mh) \\ MS_{conf_2}(mh) & , MS_{conf_2}(mh) \neq MS_{conf}(mh) \end{cases}$$

– $\forall v \in V$,

$$DS_{conf'}(v) = \begin{cases} DS_{conf_1}(v), & DS_{conf_1}(v) = DS_{conf_2}(v) \\ DS_{conf_1}(v), & DS_{conf_1}(v) \neq DS_{conf}(v) \\ DS_{conf_2}(v), & DS_{conf_2}(v) \neq DS_{conf}(v) \end{cases}$$

– $PD_{conf'} = PD_{conf_1} \cup PD_{conf_2}$
– $EX_{conf'} = EX_{conf_1} \cup EX_{conf_2}$
– $SC_{conf'} = SC_{conf_1} \cup SC_{conf_2}$

The synchrony rule is given below. *confs* is the set of target configurations obtained from the basic rule given a source configuration *conf* and an event *e*. *MergeAll* combines each configuration in *confs* into a new configuration by repeatedly combining configurations pairwise. The micro-step from c to c' by e is denoted as $c \overset{e}{\hookrightarrow} c'$.

$$\frac{\mathit{confs} = \{\mathit{conf}_{mh} | \mathit{conf} \xrightarrow{e} \mathit{conf}_{mh}\}}{\mathit{conf} \overset{e}{\hookrightarrow} \mathit{MergeAll}(\mathit{confs})}$$

Chain merge rule. The objective of the chain merge rule is to construct a macro-step, defined inductively below.

$$\frac{\mathit{conf} \overset{e1}{\hookrightarrow} \mathit{conf}'}{\mathit{conf} \overset{e1}{\rightharpoondown}_1 \mathit{conf}'}\ e1 \in \mathit{ImportedEvents}\ (1)$$

$$\frac{\begin{array}{c}\mathit{conf} \overset{e1}{\rightharpoondown}_n \mathit{conf}' \\ \mathit{conf}' \overset{e2}{\hookrightarrow} \mathit{conf}''\end{array}}{\mathit{conf} \overset{e1}{\rightharpoondown}_{n+1} \mathit{conf}''}\ (2)$$

Case (1) shows that a micro-step triggered by an imported event is the basic case. The corresponding source configuration is denoted as an *initial* configuration in the remainder of this paper. The inductive case is shown in case (2). Note that there is no restriction on how to choose $e2$ from *PD* of *conf'*. The subscript in the chain merge rule indicates the number of micro-steps from the initial configuration to the current configuration.

Discussion on design of semantic rules. The basic rule and synchrony rule are encoded in Coq as functions, because the transition of a scenario and the construction of a micro-step are deterministic. On the other hand, the chain merge rule is defined relationally because it does not specify which event to choose from *PD* to trigger the next micro-step, nor does it guarantee termination during the combination of micro-steps. To derive a computable version, which must terminate because of restrictions in Coq, we require predicates, given below, on the syntactic structure of monitor specifications, such that termination and determinism are guaranteed.

4 Towards a well-formed monitor specification

This section presents the definition of a *well-formed* monitor. Both the basic and synchrony rule are partial so we need to make sure that their application succeeds. Moreover, two vital properties for a monitor, termination and determinism, are considered. A set of predicates are proposed from which the definition of well-formedness is constructed. We prove that if a monitor satisfies these predicates, it always terminates in a final state, deterministically, which indicates that the monitor is well-formed, i.e. no runtime errors are possible. These predicates are required since only well-formed monitor specifications may be generated into executable code by Fiat.

4.1 Well-formedness predicates

Table 1 lists predicates for well-formedness, which are divided into three categories indicating which part of the execution is influenced by the predicates.

Table 1. Predicates for well-formedness

Classification	Name	Definition
Scenario level	P1	$\forall s \in S_M,\ \forall tr_1\ tr_2 \in \delta_s, qsrc_{tr_1} = qsrc_{tr_2} \wedge event_{stpEv_{tr_1}} = event_{stpEv_{tr_2}} \Rightarrow\ eventGuard_{stpEv_{tr_1}} = \neg eventGuard_{stpEv_{tr_2}}$
	P2	$\forall s \in S_M, \forall tr \in \delta_s,\ \forall e \in \Sigma_M,\ event_{stpEv_{tr}} = e\ \Rightarrow\ e \in E_s$
Micro-step level	P3	$\forall v \in V_M\ , \forall sce1\ \ sce2 \in S_M\ ,\ updateVar(v, sce1) \wedge updateVar(v, sce2) \Rightarrow E_{sce1} \cap E_{sce2} = \emptyset$
Macro-step level	P4	$\forall e\ \in\ \Sigma_M,\ eventType_e\ =\ Imported\ \vee\ eventType_e\ = Internal \Rightarrow \exists sce, sce \in S_M\ \wedge\ e \in E_{sce}$
	P5	$\forall e\ \ e1\ \ e2\ \in\ \Sigma_M,\ e1\ \neq\ e2 \wedge e\ \Uparrow_M\ e1 \wedge e\ \Uparrow_M\ e2\ \Rightarrow \neg\exists sce, sce \in S_M\ \wedge\ e1 \in E_{sce}\ \wedge\ e2 \in E_{sce}$
	P6	$\forall e\ e1 \in \Sigma_M, eventType_e = Imported\ \wedge\ e \neq e1\ \wedge\ e \Uparrow_M e1 \Rightarrow \neg\exists sce, sce \in S_M\ \wedge\ e \in E_{sce}\ \wedge\ e1 \in E_{sce}$
	P7	$\forall e \in \Sigma_M, \forall\ sce1\ \ sce2 \in S_M, raiseEv(sce1, e) \wedge raiseEv(sce2, e)\ \wedge\ sce1 \neq sce2\ \Rightarrow \neg\exists e' \in \Sigma_M, triggerSce(sce1, e')\ \wedge\ triggerSce(sce2, e')$
	P8	$\forall e \in \Sigma_M,\ sce \in S_M,\ stp \in \delta_{sce},\ noDuplicatedRaise(e, sce, stp)$
	P9	$\forall e1\ e2 \in \Sigma_M,\ \forall v \in V_M, \exists e \in \Sigma_M, noDependency(e, e1 , e2) \wedge updateVarEv(v,e1) \Rightarrow \neg updateVarEv(v,e2) \wedge \neg usedVarEv(v,e2)$

P1 and *P2* guarantee that exactly one transition is triggered for a scenario during the application of the basic rule, by an event from the alphabet for that scenario. *P3* guarantees that when applying the synchrony rule to construct a micro-step, scenarios that share the same triggering event never update the same variable. $updateVar(v, sce)$ means that variable v is updated by actions from the transitions of scenario *sce*.

A well-formed monitor guarantees that it always terminates in some final state. The definition of a final configuration is given below:

Definition 1 (Final Configuration). *A configuration conf is a final configuration if 1)* $SC_{conf} \neq \emptyset$ *and 2)* $PD_{conf} = \emptyset$.

The tricky part is that all pending events must be consumed at the end of each macro-step, i.e. there are no pending events when all the available scenarios have finished execution and that the execution of a monitor never gets stuck because of a mismatch between enabled scenarios and pending events.

P4 guarantees that all imported events or internal events can trigger execution of some scenarios. *P5* and *P6* ensure that imported or internal events that may be raised in the same macro-step cannot directly trigger execution of the same scenario. $e \Uparrow_M e1$ means that $e1$ is raised by the actions of transitions transitively triggered by *e*. *P7* and *P8* guarantee that in each macro-step, an internal event cannot be raised multiple times. *raiseEv(sce,e)* means that the actions of transitions defined in *sce* contain raising *e*. *triggerSce(sce,e)* means that *e* may transitively trigger transition of *sce*. *noDuplicatedRaise(e,sce,stp)* means that *e* can only be raised once in *stp* of *sce*.

The chain merge rule does not specify an order for the chaining of micro-steps. If a monitor is not well defined, the execution result of a macro-step could be non-deterministic. This is undesirable because we always want a deterministic verdict from a monitor given the same input. *P1* and *P2* ensure scenario-level determinism. *P5* to *P8* also prevent some behaviors that may lead to non-determinism. We define a proposition *noDependency(e,e1,e2)* $\overset{\text{def}}{=}$ $\mathit{eventType}_e = \mathit{Imported} \wedge e \Uparrow_M e1 \wedge e \Uparrow_M e2 \wedge \neg e1 \Uparrow^e_M e2 \wedge \neg e2 \Uparrow^e_M e1$. This means that $e1$ and $e2$ may be raised in the macro-step triggered by imported event e, and that during this macro-step, $e1$ can not transitively raise $e2$, and vice versa. *P9* guarantees that updating a state variable is mutually exclusive. *updateVarEv(v,e)* and *usedVarEv(v,e)* respectively mean that v cannot be updated and used in any actions transitively triggered by e.

We use the notation $Pi(M)$ to represent that a monitor M satisfies predicate *Pi*. A well-formed monitor satisfies the nine predicates defined above, *Well-formed(M)* $\overset{\text{def}}{=} \bigwedge_{1 \leq i \leq 9} Pi(M)$.

4.2 Proof of termination and determinism

Given a monitor that is well-formed, and which starts execution with a well-typed imported event, we can now prove that it can always reach a final state within a limited number of micro-steps, as described in Theorem 1 below:

Theorem 1 (Termination). *Given a well-formed monitor M, two of its configurations* conf_M *and* conf'_M *and an imported event e, if* $\mathit{conf}_M \xrightarrow{e}_n \mathit{conf}'_M$ *and M cannot take any micro-step from* conf'_M*,* conf'_M *is a final configuration and n is equal to or less than* $|S_M|$*.*

To prove this theorem, we need to first prove that the number of micro-steps taken within a macro-step is bounded. Because each scenario can only transition once during each macro-step, and at least one scenario executes in each micro-step, the number of micro-steps to be taken is bounded by the number of scenarios of the monitor. So we first prove that $|SC_{conf}|$ strictly increases in a micro-step.

Lemma 1 (Increase of SC). *Given two configurations conf conf′ and an event e, if* $\mathit{conf} \overset{e}{\hookrightarrow} \mathit{conf}'$*, then* $|SC_{conf}| < |SC_{conf'}|$*.*

With Lemma 1 and the fact that SC_{conf_M} is a subset of $|S_M|$, we can prove that the number of micro-steps taken by a well-formed monitor in a macro-step is bounded by the number of scenarios:

Lemma 2 (Up-bound of micro-steps). *Given a well-formed monitor M, two of its configurations* conf_M *and* conf'_M *and an imported event e, if* $\mathit{conf}_M \xrightarrow{e}_n \mathit{conf}'_M$*, then* $n \leq |S_M|$*.*

Next we need to prove that macro-step has the *progress* property, which guarantees that a well-formed monitor cannot be stuck in a non-final state:

Lemma 3 (Progress). *Given a well-formed monitor M, two of its configurations $conf_M$ and $conf'_M$ and an imported event e, if $conf_M \xrightharpoonup{e}_n conf'_M$ and $conf'_M$ is not a final configuration, then M can take a micro-step on all of its pending events from $conf'_M$.*

With the three core lemmas presented above, and other auxiliary lemmas, Theorem 1 can be proved. With this theorem, we can always pick a terminating implementation of the relational semantic rules during the refinement step.

Deterministic execution of a macro-step is represented by the theorem below:

Theorem 2 (Determinism). *Given a well-formed monitor M, if $conf_M \xrightharpoonup{e} conf'_M$, $conf_M \xrightharpoonup{e} conf''_M$ and both $conf'_M$ and $conf''_M$ are final configurations, then $conf'_M = conf''_M$.*

This theorem is proved using the idea of Newman's lemma [9]. First, we prove the *diamond* lemma defined below:

Lemma 4 (Diamond). *Given a well-formed monitor M, if $conf_M$ is an initial configuration or there exists a configuration oconf such that $oconf \xrightharpoonup{e} conf_M$, and $conf_M \overset{e1}{\hookrightarrow} conf1_M$ and $conf_M \overset{e2}{\hookrightarrow} conf2_M$, then there exists a configuration $conf'_M$ such that $conf1_M \overset{e2}{\hookrightarrow} conf'_M$ and $conf2_M \overset{e1}{\hookrightarrow} conf'_M$.*

Then, by induction on the number of micro-steps to be taken by two transition chains, we can prove the *confluence* lemma:

Lemma 5 (Confluence). *Given a well-formed monitor M, if $conf_M \xrightharpoonup{e} conf1_M$, $conf_M \xrightharpoonup{e} conf2_M$, there exists a configuration $conf'_M$ such that $conf1_M \hookrightarrow_* conf'_M$ and $conf2_M \hookrightarrow_* conf'_M$.*

Transition $\hookrightarrow_*$ represents multiple micro-steps. Lemma 5 ensures that if an initial configuration *conf* can transition into two non-final configurations $conf_1$ and $conf_2$, then they can always transition back to the same configuration. Using Lemma 5 and the fact that a final configuration cannot take any micro-step, Theorem 2 can be proved.

5 Refinement of a Monitor Specification using Fiat

This section presents how to generate correct-by-construction code from a declarative ADT using Fiat. Fig 2 gives an overview of the code generation process. The initial ADT describes the basic behavior of monitors in a declarative way using semantic rules defined in the previous section. Then, the ADT is refined by proving a "sharpening" theorem, wherein the representation type, constructors and methods of the ADT are refined. The refinement of methods involves picking a specific implementation and proving that $\supseteq_r$, introduced in Section 2, is preserved between the specification and the implementation. The implementation is parameterized by a specific monitor definition given a starting state and proof of well-formedness of that monitor. Haskell code can then be extracted from this definition.

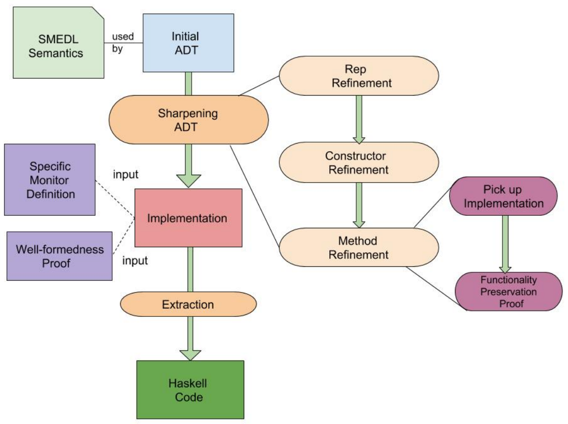

Fig. 2. Code generation process

5.1 Definition of an ADT

Basically, the monitor ADT describes the common process of handling imported events using the semantic rules defined in the previous section. The definition of this ADT is given below.

```
Definition confSpec : ADT _ := Def ADT {
  rep := configuration M,
  Def Constructor0 newS : rep := { c : configuration M | readyConfig c },,
  Def Method1 processEventS (r : rep) (e: raisedEvent | raisedAsImported  M e) :
      rep * list raisedEvent :=
      { p : rep * list raisedEvent
    | exists conf' econf,
        chainMergeTrans r conf' econf (` e) (fst p) (snd p) }
}.
```

The configuration of a given monitor M is used as the representation type for the ADT. Instead of constructing a concrete value, Constructor *newS* specifies that the starting state of a monitor should be a *ready* configuration. A ready configuration has empty sets for PD, EX and SC, indicating that the monitor is ready to receive an imported event for the next macro-step. The method *processEventS* specifies the non-deterministic action of taking a macro-step. The first parameter r represents the current ready state of the monitor and the second parameter e is the imported event triggering the macro-step. The return value is a tuple of a ready configuration that reflects the updated state of the monitor after the macro-step and a list of raised exported events. The semantic rules from previous sections were defined in a relational way to conveniently specify this method, since relations easily model non-deterministic functions. To adapt the chain merge rule to the interface of *processEventS*. *chainMergeTrans* is defined below:

```
Definition chainMergeTrans {M : monitor} (conf conf' econf: configuration M)
(e: raisedEvent) (rconf: configuration M)  (events: list raisedEvent)  : Prop :=
   configTrans conf conf' /\
   chainMerge conf' econf e /\
   finalConfig econf  /\
   configTransRev econf rconf /\
   events = EX econf.
```

configTrans conf conf' represents the transformation from ready configuration *conf* to initial configuration *conf'*; *chainMerge conf' econf e* is the Coq definition of $conf' \xrightharpoonup{e} econf$ with the number of steps taken omitted; and *configTransRev* represents the transformation from *econf* to a new ready configuration *rconf*. *events* is the set of exported events raised in this macro-step.

5.2 Refinement process

Refinement by Fiat requires proving the theorem *FullySharpened(confSpec M)*, parameterized over some monitor definition *M*. The implementation is wrapped in the proof term of the theorem. The first step refines the representation type. In this paper, we choose the same representation type—the configuration of monitor *M*—in the implementation. As a result, the *abstraction relation r* is plain equality. Constructor *newS* is refined by choosing a ready configuration *conf* for monitor *M*, given by the starting state of monitor *M*. Just like parameter *M*, *conf* also needs to be provided to generate a concrete, executable monitor. To refine method *processEventS*, we need to provide a deterministic function that preserves the semantics of applying the chain merge rule. Preservation of the specification's semantics for this function is given by the lemma below:

```
Lemma ProcessEventRefined M (C : configuration M) (W : Wellformed M)
      (Cor:readyConfig C) (e: raisedEvent) (P : raisedAsImported M e) :
 refine { p : configuration M * list raisedEvent
        | exists conf' econf, chainMergeTrans C conf' econf e (fst p) (snd p) }
        (ret (macroStepReadyFinal W Cor P
                                 (length (S M)))).
```

macroStepReadyFinal is a function which takes a ready configuration C and returns a new ready configuration and list of exported events. In this paper, we choose a straightforward implementation: a fixpoint function that picks the first event from *PD* of the current configuration to trigger the next micro-step. Note that in the Coq definition, we use a list to represent the set, and due to the predicates establishing well-formedness, *PD* can never have duplicate events. The number of times the semantic function gets invoked is bounded by the number of scenarios in *M*. Provided that *M* is well-formed, it is guaranteed that the resulting configuration is a final configuration. The lemma *ProcessEventRefined* establishes that the return value is a subset of the results obtained by applying *chainMergeTrans* used in the original ADT. From the proof term of the theorem, an executable version of *processEventS* can be obtained.

It is worth noting that, the semantics of SMEDL can be directly expressed as a Coq function for generating the Haskell code by native Coq. But through Fiat, we can refine from the declarative SMEDL semantics to a more efficient implementation by changing the data structure for configuration, handling pending events more wisely, etc. Moreover, refinement can be conducted in a more mechanical and extensible way in Fiat than using native Coq.

6 Case Study

A general event processing function is generated by refinement, parameterized by: a specific monitor specification, its well-formedness proof and a starting, ready state for that monitor. Therefore, to obtain a correct-by-construction monitor, one needs to 1) write a monitor definition M; 2) prove that M is well-formed; and 3) specify a starting state. A Haskell program may then be extracted, from which a monitor is implemented by adding glue code to receive events from the target system. This section uses a real-world monitoring requirement to illustrate the usability of this method.

SMEDL specification. The monitoring requirement comes from a known vulnerability(CVE-2017-9228)[2] in Oniguruma v6.2.0 [10], which is related to incorrect parsing of regular expressions, resulting in a crash due to access of an uninitialized variable. Based on a high-level specification, and agnostic of the specific vulnerability, a SMEDL monitor is constructed to detect this violation of the specification and raise an alarm. The specification is based on a part of the regular expression grammar concerning *character classes*. The parsing can be described as the state machine in Fig 3, where transition labels are omitted for clarity. Transitions in the state machine are triggered by tokens read by the parser and guarded with additional conditions.

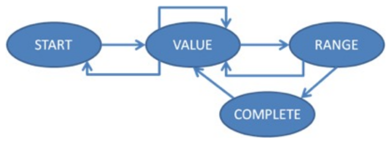

Fig. 3. State machine of parsing character class

Part of the SMEDL specification (denoted as *parseCC*) is given below. To simplify the presentation, we concentrate only on one guard condition, which states that the VALUE state cannot be recursively entered (i.e., from the START state) while a class is still being processed. We refer to this condition below as *in_class* being equal to 1. Scenario *main* records transitions of the parser state machine affected by the code; they are triggered by events that correspond by changes to state variables in the code. Scenario *check_class* determines the value of the state variable *in_class* by receiving the imported events *in_class* and *out_class*. The specification can be directly mapped to an AST definition in Coq.

```
object parseCC
    state
        int in_class = 0;
    events
        imported inClass();//enter next_value_class
        imported outClass();//exit next_value_class
        imported state_to_start();//state is set to START
```

[2] https://nvd.nist.gov/vuln/detail/CVE-2017-9228

```
        imported state_to_value();//state is set to VALUE
        imported state_to_range();//state is set to RANGE
        imported state_to_complete();//state is set to COMPLETE
        exported error(int);
    scenarios
       main:
           START -> state_to_value() when (in_class != 1) -> VALUE
           START -> state_to_value() when (in_class == 1)
           {raise error(0);} -> START
           VALUE -> state_to_value() -> VALUE
           VALUE -> state_to_range() -> RANGE
           VALUE -> state_to_start() -> START
           ...

       check_class:
          idle -> in_class() when (in_class == 0)
                {in_class = 1;} -> idle
          idle -> out_class() when (in_class == 1)
                {in_class = 0;} -> idle
```

Proof of well-formedness. Proving the well-formedness of a monitor seems hard because there are nine sub predicates needed to be proved and type correctness needs to be checked. However, we have implemented decision procedures to check whether a monitor satisfies *P1* to *P4*. Rest of them can be proved using the auxiliary lemmas and tactics. The LOC for the proof is less than 1k of Gallina and Ltac code. The time for proving well-formedness of *parseCC* is estimated to be about 30 minutes for a user with basic experience of Coq.

Construction of the Haskell monitor. The core building block of a *parseCC*-based monitor is given below. *processEventS* is the general event handling function refined from the Fiat ADT. The Parameter r contains the information to be used by *parseCC*: the proof of well-formedness (denoted as *Well_ParseCC*) and a starting state. Parameter e is the imported, triggering event for *parseCC*.

```
Definition parseCC_processEvent (r : ComputationalADT.cRep
            program Well_ParseCC configuration1_ready)
           (e: raisedEvent | raisedAsImported  parseCC e) :=
  processEventS r e.
```

Coq provides the ability to extract Coq definitions to a Haskell program. The monitor is constructed by adding glue code for receiving events from the target system. We compile the Haskell code into an object file and expose two functions to be instrumented into the target program. The type signature of these two Haskell functions are given below:

```
cInitialRep :: IO (Ptr ())
cHandleImported :: CString -> Ptr () -> IO (Ptr ())
```

Both functions rely on the extracted Haskell code. *cInitialRep* provides a starting state for the monitor. *chandleImported* takes the name of an imported event, and the current state of the monitor, and returns a new state with any exported events printed out. The target system is responsible for recording this state update transparently. Using the GHC compiler, both an object file and a C header file are generated. The header file contains the C API of the two functions defined above, which are called in the source code of Oniguruma. When an incorrect transition occurs in the library, an alarm is raised and printed to the screen.

The LOC for the automatically extracted code is about 6k lines but only about 10% of the code depends on the definition of a monitor. For another example monitor with 6 scenarios and 16 transitions, the LOC of the part depending on the monitor definition is less than 1k lines. Thus, scalability would not be an issue.

The difficult part of deriving a monitor is its proof of well-formedness, which can be simplified using the provided decision procedures and tactics. The other steps are easily implemented using common procedures. The methodology presented in this paper provides an straightforward way to implement correct-by-construction monitors.

7 Related Work

We summarize a representative selection of related work in three categories: 1) formal semantics for RV; 2) mechanization of semantics for state-machine-based formalisms and 3) case studies of using the Fiat framework.

The semantics of temporal logic and traces used for RV have been widely studied [11,12,13,14,15]. There has been a lot of work related to describing the semantics of LTL/MTL using automata-like formalisms. Giannakopoulou and Havelund present a technique translating LTL formulae into FSM to monitor program behavior [16]. Drusinsky presents TLCharts, a formalism resembling Harel statecharts while supporting the specification of nondeterministic, temporal properties described in MTL or LTL inside a statechart specification [17]. This semantics is described using Equivalent Non-Deterministic Automaton (ENFA). Roşu and Havelund propose a method for rewriting LTL formulas into binary transition tree finite state machines for online monitoring [18]. Several RV tools support using FSM to specify properties [19,20,21,22,23,24]. However, little work has been done on mechanizing semantics of DSLs for RV. Our work shows that the semantic mechanization is a necessary foundation for generating correct monitors.

In [25], Paulin-Mohring presents the model of timed automata in Coq for specifying and verifying of telecommunication protocols. Kammüller and Helke [26] formalize the semantics of Statecharts [27] using Isabelle/HOL [28]. In AADL (Architecture Analysis and Design Language) [29] , the thread model and mode change are represented using automata. Yang et al. [30] propose a machine-checked transformation of a subset of AADL into TASM (Timed Abstract State Machine [31]). The main purpose of defining formal semantics in these studies is to prove properties of formal models. Although the formal semantics of SMEDL can be used to prove properties of SMEDL or a monitor, the primary objective of our work is code generation. Particularly, the semantic rules have been designed to be conveniently integrated into a Fiat specification.

Correct-by-construction implementation generation using refinement has been well studied [32,7,33,34,35,36]. For instance, Event-B [37] refines an abstract transition system into a more concrete one by adding transitions and states. Fiat is a more general tool, suitable for the refinement objective of SMEDL. Fiat provides flexible support for refinement of libraries for different functionality domains. Delaware et al. [3] illustrate an example of using Fiat to synthesize query structures. Wiegley and Delaware [6] use Fiat to generate efficient and correct implementations of a subset of the bytestring library in Haskell. Chlipala et al. [5] present the development of a simple packet filter in Fiat. The ADT for SMEDL is not as complicated as the case studies listed above, but it is an initial work using Fiat to refine a state-machine-based DSL. The semantic rules and ADT design presented in this paper offer guidance for applying Fiat to generate code for other state-machine-style DSLs.

8 Discussion and Conclusions

We have presented a method for deriving correct-by-construction monitor code using the Fiat framework. An operational semantics of SMEDL is designed using Fiat to describe the essential behavior of any monitor. Using the mechanisms provided by Fiat, this ADT is then refined into executable monitor code while preserving those semantics. We have also proposed a well-formedness predicate on monitor structures and proved that if a monitor is well-formed, it can always terminate deterministically in a final state after reacting to any imported event.

One concern of using formal techniques is the manual effort involved in proof work. In our development, proofs are divided into two parts: One part includes proofs used during the refinement process, and auxiliary tactics and decision procedures for proving the well-formedness of any monitor; the other part is the proof of well-formedness for a particular monitor. The raw LOC in Coq for the first part is about 30k lines. However, to apply the technique, users only need prove well-formedness of their particular monitor, which is not labor-intensive given the help of auxiliary tactics and lemmas. Therefore, we assert that generating correct runtime monitors using a proof assistant is a feasible task.

One main avenue of future work would be to improve the definition of well-formedness, and implement more auxiliary tactics for better usability. It is also worth exploring more efficient implementation of the semantic rules in order to generate more optimized monitor code.

References

1. Sokolsky, O., Havelund, K., Lee, I.: Introduction to the special section on runtime verification. Software Tools for Technology Transfer **14**(3) (2012) 243–247
2. Zhang, T., Gebhard, P., Sokolsky, O.: SMEDL: Combining synchronous and asynchronous monitoring. In: International Conference on Runtime Verification, Springer (2016) 482–490
3. Delaware, B., Pit-Claudel, C., Gross, J., Chlipala, A.: Fiat: Deductive synthesis of abstract data types in a proof assistant. In: ACM SIGPLAN Notices. Volume 50., ACM (2015) 689–700

4. The Coq Development Team: The Coq Proof Assistant Reference Manual
5. Chlipala, A., Delaware, B., Duchovni, S., Gross, J., Pit-Claudel, C., Suriyakarn, S., Wang, P., Ye, K.: The end of history? using a proof assistant to replace language design with library design. In: SNAPL'17: 2nd Summit on Advances in Programming Languages. (2017)
6. Wiegley, J., Delaware, B.: Using Coq to write fast and correct haskell. In: Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, ACM (2017) 52–62
7. Hoare, C., Hayes, I., He, J., Morgan, C., Sanders, J., Sorensen, I., Spivey, J., Sufrin, B.: Data refinement refined. (1985)
8. Cheng, K.T., Krishnakumar, A.S.: Automatic functional test generation using the extended finite state machine model. In: Proceedings of the 30th international Design Automation Conference, ACM (1993) 86–91
9. Newman, M.H.A.: On theories with a combinatorial definition of" equivalence". Annals of mathematics (1942) 223–243
10. Oniguruma contributors: Oniguruma. `https://github.com/kkos/oniguruma` Accessed: 2018-03-27.
11. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: International Workshop on Verification, Model Checking, and Abstract Interpretation, Springer (2004) 44–57
12. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., De Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: ACM SIGPLAN Notices. Volume 40., ACM (2005) 345–364
13. Barringer, H., Rydeheard, D., Havelund, K.: Rule systems for run-time monitoring: from Eagle to RuleR. Journal of Logic and Computation **20**(3) (2010) 675–706
14. Chen, F., Roşu, G.: Parametric trace slicing and monitoring. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer (2009) 246–261
15. Lee, I., Kannan, S., Kim, M., Sokolsky, O., Viswanathan, M.: Runtime assurance based on formal specifications. Departmental Papers (CIS) (1999) 294
16. Giannakopoulou, D., Havelund, K.: Automata-based verification of temporal properties on running programs. In: Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on, IEEE (2001) 412–416
17. Drusinsky, D.: Semantics and runtime monitoring of tlcharts: Statechart automata with temporal logic conditioned transitions. Electronic Notes in Theoretical Computer Science **113** (2005) 3–21
18. Roşu, G., Havelund, K.: Rewriting-based techniques for runtime verification. Automated Software Engineering **12**(2) (2005) 151–197
19. Colombo, C., Pace, G.J., Schneider, G.: Dynamic event-based runtime monitoring of real-time and contextual properties. In: International Workshop on Formal Methods for Industrial Critical Systems, Springer (2008) 135–149
20. Havelund, K.: Runtime verification of C programs. Testing of Software and Communicating Systems (2008) 7–22
21. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. International Journal on Software Tools for Technology Transfer **14**(3) (2012) 249–289
22. Luo, Q., Zhang, Y., Lee, C., Jin, D., Meredith, P.O., Şerbănuţă, T.F., Roşu, G.: RV-Monitor: Efficient parametric runtime verification with simultaneous properties. In: International Conference on Runtime Verification, Springer (2014) 285–300

23. Chen, Z., Wang, Z., Zhu, Y., Xi, H., Yang, Z.: Parametric runtime verification of C programs. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer (2016) 299–315
24. Reger, G., Cruz, H.C., Rydeheard, D.E.: MarQ: Monitoring at runtime with QEA. In: TACAS. (2015) 596–610
25. Paulin-Mohring, C.: Modelisation of timed automata in Coq. In: International Symposium on Theoretical Aspects of Computer Software, Springer (2001) 298–315
26. Kammüller, F., Helke, S.: Mechanical analysis of UML state machines and class diagrams. In: IN THE PROC. OF WORKSHOP ON PRECISE SEMANTICS FOR THE UML. ECOOP2000, Citeseer (2000)
27. Harel, D.: Statecharts: A visual formalism for complex systems. Science of computer programming **8**(3) (1987) 231–274
28. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic. Volume 2283. Springer Science & Business Media (2002)
29. Frana, R., Bodeveix, J.P., Filali, M., Rolland, J.F.: The AADL behaviour annex–experiments and roadmap. In: Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on, IEEE (2007) 377–382
30. Yang, Z., Hu, K., Ma, D., Bodeveix, J.P., Pi, L., Talpin, J.P.: From AADL to timed abstract state machines: A verified model transformation. Journal of Systems and Software **93** (2014) 42–68
31. Ouimet, M., Lundqvist, K., Nolin, M.: The timed abstract state machine language: An executable specification language for reactive real-time systems. RTNS?07 (2007) 15
32. Dijkstra, E.W.: A constructive approach to the problem of program correctness. BIT Numerical Mathematics **8**(3) (1968) 174–186
33. Srinivas, Y.V., Jüllig, R.: Specware: Formal support for composing software. In: International Conference on Mathematics of Program Construction, Springer (1995) 399–422
34. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to hopcroft's algorithm. In: International Conference on Interactive Theorem Proving, Springer (2012) 166–182
35. Cohen, C., Dénès, M., Mörtberg, A.: Refinements for free! In: International Conference on Certified Programs and Proofs, Springer (2013) 147–162
36. Lammich, P.: Refinement to Imperative/HOL. In: International Conference on Interactive Theorem Proving, Springer (2015) 253–269
37. Abrial, J.R., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: Application to event-b. Fundamenta Informaticae **77** (2007) 1–28