Centre Fédéré en Vérification

Technical Report number 2004.40

Class-Level Behavioral Modeling and Synthesis

Yves Bontemps, Patrick Heymans, Germain Saval, Pierre-Yves Schobbens,
Jean-Christophe Trigaux

This work was partially supported by a FRFC grant: 2.4530.02

http://www.ulb.ac.be/di/ssd/cfv

Class-Level Behavioral Modeling and Synthesis

Yves Bontemps[*], Patrick Heymans, Germain Saval,
Pierre-Yves Schobbens, Jean-Christophe Trigaux[†]
University of Namur
Computer Science Department
Belgium
{ybo,phe,gsa,pys,jtr}@info.fundp.ac.be

Abstract

When modeling the behavioral requirements of object-oriented distributed systems, one has to take class-level scenarios into account. Those describe interactions that apply to all/some instances of some class. Current scenario-based notations fall short on this problem. They remain at instance level and suitable only for modeling the behavior of a particular population. State machines are often used later in the analysis to factor scenarios classwise. However, when dealing with class-level behavior, state machine models get unnecessarily awkward and implementation-oriented, incorporating iterations over instances, for example. This paper brings two contributions solving this crucial issue. First, we propose an obvious extension of current scenario and state machine languages, already envisioned by Harel back in 1984: all notations get extended with universal and existential quantifiers. Second, we upgrade classical synthesis algorithms dealing with class-level notations. This extension preserves properties of instance-level algorithms.

1. Introduction

Typically, in distributed object system development, software engineers start designing a system by writing scenarios. They describe some examples of behavior, then add up more and more such scenarios, which they eventually have to combine. Once all scenarios have been combined, they have a global and complete description of object interactions, hopefully. Their ultimate goal is to build an executable distributed program for every class, which will actually exhibit the behavior prescribed by the scenarios.

In line with the OMG (Object Management Group)'s MDA (Model Driven Architecture) initiative [21], many authors have proposed to automate the step from scenarios to state machines [2, 19, 17, 16, 24, 27]. Scenarios are usually expressed with UML Sequence Diagrams [22] or Message Sequence Charts (MSCs) [25]. These offer a "one story for all objects" perspective while the implementation of a distributed system needs "all stories for one object" [12], i.e. one executable description for each class. State machines (e.g. UML State Diagrams [22]) are an obvious choice for this purpose. Transforming one into the other is basically a classwise factorization of the behavior expressed in the scenarios. In MDA terms, this is a fully automated PIM (Platform Independent Model) to PIM transformation. Further transformations into a PSM (Platform Specific Model) and then to code are not in the scope of this paper, and are handled by existing tools.

In previous papers [5, 9], we have proposed synthesis and verification algorithms for LSCs (Live Sequence Charts) [10], a variant of MSCs enriched with message abstraction and modalities. Subsequently, we have optimized these algorithms to better cope with the state explosion problem [6]. In this paper, we upgrade our notations and tools to handle class-level behaviors of an unknown number of runtime object instances, as is the case in the vast majority of object systems. To do this, we need to add explicit class-level constructs to existing notations. Our extension of Sequence Diagrams or MSCs is called QHMSCs (Quantified High-Level Message Sequence Charts). QHCSMs (Quantified Hierarchical Concurrent State Machines) is our extension of State Diagrams. As we will show, these allow to formally and concisely specify the behavior of object populations that are not known in advance.

In Section 2, we describe the syntax and semantics of our class-level-enhanced source (2.3) and target (2.2) models. We also state the basic hypotheses on the distributed object systems we consider (2.1). Section 3 is devoted to synthesis. It opens with a formal problem statement (3.1) and goes

[*]Research Fellow of the FNRS (Belgian National Fund for Scientific Research)

[†]Work Supported by Walloon Region and FSE in FIRST Europe Objective 3 Project PLENTY EPH3310300R0462 / 215315.

on with a description of the algorithms (3.2) and a proof of their semantic-preserving properties. Our algorithms are then applied to an example extracted from NASA's CTAS system [28, 7] in Section 3.3. The papers ends with related works (Section 4), conclusions and future works (Section 5).

2. Models

In this section, we present the modeling languages we use. The syntax definition delineates what models are correctly formed in our languages. The semantics assigns a meaning to every syntactically correct model.

We start by introducing the semantic domain which defines the types of systems that are modeled with those languages. Our introduction is first presented informally, then mathematized.

2.1. Distributed Object Systems

We assume that we deal with very general *distributed object systems*, that is a finite set of objects. Every object is an *instance* of a class and has an identifying name. Classes group similar objects.Yet, an object can play several roles, thanks to *inheritance*: classes are arranged in an inheritance hierarchy. If some class A inherits from class B, then, all instances of class A are also instances of class B. Every object has an *actual class*, which is its most precise class. An object system exhibits a certain behavior: objects pass messages to each other. Those messages need not be treated immediately: every object owns a *message pool*, in which other objects drop messages. An object can pick up a message from its pool and, as a reaction to this message, can decide to send some messages to other instances in the system. Picking up a message in its pool is a *receive event*, while dropping a message in a pool is a *send event*. An execution of the object system is a sequence of interactions of this form: an object *receives* a message and reacts by *sending* messages. The behavior of an object system is the set of all executions of this system.

Formally, we are given a countable set of class names, $\mathbb{C}$, with an inheritance partial ordering, $A \sqsubseteq B$ meaning A inherits from B and a countable set of object names $\mathbb{O}$. A population is a function $p : \mathbb{O} \to \mathbb{C}$, assigning its *actual class* to every object. Message names are taken from a set Γ. A message contains three elements: its name, the name of its sender and the name of its receiver. Formally, messages are thus $\mathbb{M} = \mathbb{O} \times \Gamma \times \mathbb{O}$. An event is either "receiving message m", denoted $?m \in ?\mathbb{M}$ or sending m, $!m \in !\mathbb{M}$. The events of an object o, $\Sigma(o)$ are of the form $!(o, \gamma, o')$ and $?(o', \gamma, o)$. The set of all events is denoted $\Sigma = \Sigma_? \uplus \Sigma_!$. An execution is a finite sequence of events: $e_0 \dots e_n \in \Sigma^*$.

We have just described a particular object system, with a fixed number of instances of particular names and a certain behavior. However, practically, we want to express properties about *all* object systems of a certain kind. We will thus take into account all possible object populations.

The set of all populations is $\mathbb{P}$. Our semantic domain is therefore $\mathbb{P} \to 2^{\Sigma^*}$, i.e. we have to assign a set of executions to every possible population.

2.2. State Machines

As an example, consider the alarm system presented as a Class Diagram [22] in Fig. 1. Remark that, as announced, there is an infinity of possible populations, with many sensors, alarms, guards and input devices, having different names. The behavior of *all* instances of `Alarm` is described by the state machine of Fig. 2. Thus, to every class, we associate one state machine. For a given population, this state machine will be *instantiated*, i.e. copied as many times as there are instances of the considered class, and its transition labels will be renamed, with its owner's name.

Our flavour of state machines is based on Harel's Statecharts [11]: it adds concurrency, hierarchy *and* quantification to plain old state machines. We call them *Quantified Hierarchical Concurrent State Machines*, QHCSM, for short. We avoid calling these machines Statecharts, because, even though they *are* inspired by Statecharts, they miss some features of this language, add new constructs and do not follow their semantics. A QHCSM is made of states, represented as rounded boxes. States are linked by arrows, called *transitions*, which are labeled by elements of the form "event-guard-actions" ($e[g]/a$), all of them being optional. When the object is in the source state of such a transition, message e is in its pool and the guard g is true, the object can follow the transition to the target state, sending all messages of a. For instance, when "alarm" is `engaged` and some of its sensors has sent an alert, it must move to the state in which it will deal with this alarm and ring its bell.

States can contain in turn state machines. They can even contain several machines at once; in this case, these are executing concurrently. We describe informally the semantics of our state machines. A formal Structured Operational Semantics can be found in [8]. A configuration of a state machine is a marking of the diagram such that

1. Exactly one top-level state (say, q) is in the configuration;
2. For every sub-machine M contained in q, the marking defines a configuration of M.

When the configuration contains the source state of a transition labeled by $e[g]/a$, if message e is in its pool and the guard g is true, the object can follow the transition to the

target state, say q', sending all messages of a. For instance, when "alarm" is `engaged` and some of its sensors has sent an alert, it rings its bell and goes to the state in which it will deal with this situation. The new configuration is then computed by first removing the source state of the transition and all its sub-states, and then adding the target state with all the initial configurations of its sub-machines.

When a state is entered, its embedded state machines are entered, too. Default initial state are pointed to by small arrows originating from a black dot, e.g. `off` is an initial state. Circled dots denote final states. When the execution of a sub-machine reaches such a state, it is considered terminated. Some special transitions, named *completion transitions*, are taken as soon as *all* sub-machines of some state are finished. Syntactically, they are represented as unlabeled transitions, starting from the border of a state (not from exception gates, see below).

States can be labeled by *invariants*, as in Timed Automata [1] or UML Protocol State Machines [22]. This invariant must hold as long as a configuration contains that state. Syntactically, an invariant is a bracketed condition written within a state, see φ in the rule labeled *sm*$(\varphi \implies M)$ in Fig. 3, for instance.

If no transition is available at top-level, then transitions can be taken in sub-machines. Actually, all enabled transitions in sub-machines are executed. We thus follow the priority scheme of STATEMATE [14].

We add the possibility to quantify state machines, either universally or existentially over instances of a class. A quantification, over a class C, is an expression of the form "for all $x : C : \phi$" or "for some $x : C : \phi$". It introduces a new variable x *bound* in the overall formula, but *free* in ϕ. Universal quantifications means "as many concurrently executing copies of the quantified state machine as there are instances of class C fulfilling ϕ". Existential quantification is nothing but a choice of some instance of C satisfying ϕ. For a given population, it is thus possible to transform a QHCSM into a plain state machine, without quantification. These extensions were already envisioned by Harel in his seminal 1984 paper [11]. For instance, in Fig. 2, in the middle-left state, the alarm sends message "disable" to all its sensors, to deactivate them. Note that this does *not* impose any iteration order on `sensors`, leaving the behavioral specification purposely abstract. When designing the system, time is not ripe for making a decision about how this should be implemented. When all sensors will have been disabled, the alarm will move to state `off`. The transition to `off` is a completion transition which is followed as soon as all component state machines are over.

An example of existential quantification appears in the lowest state of Fig. 2. The alarm picks some of its guards (among the guards on duty) and pages him. If the guard does not acknowledge the call (i.e. sends back `nack`), this state is exited and reentered, causing a guard to be chosen, again. Otherwise, this sub-machine terminates correctly, and waits for its concurrent state machines to terminate too, before moving back to `off`.

Finally, we make use of "exceptions". Exceptions are just a convenient way of structuring inter-level transitions, which are forbidden in QHMSC, thus easily preserving compositionality [26]. Exception gates are represented, in UML style, by a small circled cross, on the border of a state. When such a gate is reached, the whole sub-machine is interrupted and the transition is prolonged to the upper level. For instance, in the upper-right state of Fig. 2, as soon as *some* sensor answers that it is not working properly, the whole testing protocol is aborted and all sensors get disabled. Exception transitions have a higher priority than inner transitions.

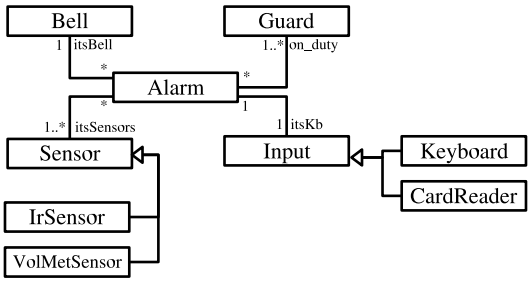

Figure 1. Alarm System: Class Diagram

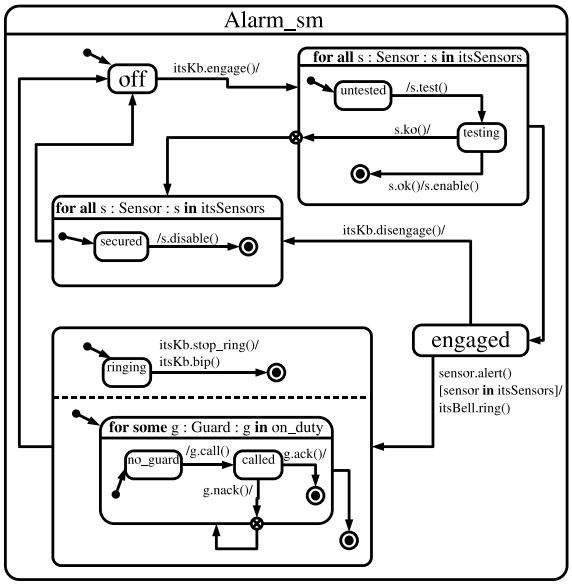

Figure 2. Alarm System: State Machine for class "Alarm"

2.3. Scenarios

The idea of a class-level state-based notation is thus natural, intuitive and also easy to define formally. We must now also find a natural notation for scenarios, which are usually expressed with Message Sequence Charts (MSC) or, within UML, by Sequence Diagrams (SD). We thus extend these notation with class-level constructs. In UML, vertical lifelines must be topped by a rectangle containing an instance identifier, underlined (see Fig 5, left). We propose to introduce also variable declarations. They come in two forms: universal, drawn as a solid rectangle (see c::Client in Fig 7) and existential, drawn as a dotted rectangle (see d::DataSource in Fig 7), following [20]. We allow them to occur not only at the top of a chart, but also at lower places. The introduced variable will depend only on variables for which a *dependency arrow* is drawn. The dependency arrow must always go down, so that dependency cycles are forbidden. It is often the case that a dependency is submitted to the existence of a given relation (declared in the class diagram): as a shorthand, we allow to mention the relation directly on the dependency arrow (see `connected`, `mirrors` in Fig 7). To know where the scope of the variable ends, we introduce *termination points* drawn as greyed circles. Note that termination points just terminate the scope of the variable, but not the existence of the instance. Symmetrically, variable declarations do not create an instance but just binds some/all instances with the name of the variable. Declaration and termination points must be well-balanced, i.e. located in the same box.

We also devise a textual notation, more convenient for theory.

$$\begin{array}{rcl} M & ::= & B | M_1 \parallel M_2 | M_1 ; M_2 | M_1 + M_2 | M^* \\ & | & \forall x : C \, (R(t,x) \implies (M)) \\ & | & \exists x : C \, (R(t,x) \wedge (M)) \end{array}$$

When no relation R is noted on a dependency arrow from t to x, it is simply taken to be true. t is either a variable or an instance. We do not detail the textual syntax of a basic chart B here, but it simply gives a labeled partial order between events [25]. The two syntaxes are intertranslatable.

3. Synthesis

3.1. Problem

At design time, QHMSCs are supposed to model the complete behavior of the future system. The executions described by the QHMSC shall match exactly the actual executions of the distributed system. Abstractly, for some population p, if we identify the future system with its behavior (i.e. its set of executions) $B \subseteq \Sigma^*$, the best implementation (wrt some QHMSC M) is achieved when $B = [\![M]\!]p$. However, B must be a distributed implementation. In particular, it means that objects in B base their decision of performing their own actions on their local view of the execution only. The local view of an execution (for some object o) is simply the projection of the trace onto o's events. Projecting a word w on an alphabet A (denoted $w|_A$) means discarding all events of w that do not belong to A. Therefore, in a distributed system, an object must act the same way after u or u' if it cannot distinguish between them.

Definition 1 (Undistinguishable ($\sim_o$)) *u, u' are undistinguishable for object o (written $u \sim_o u'$) iff $u|_{\Sigma(o)} = u'|_{\Sigma(o)}$.*

Definition 2 (Distributed Behavior) *Formally, a behavior B is* distributed *iff for every $u, v \in \Sigma^*$ and $a \in \Sigma(o)$,*

$$uav \in B \iff \forall u' \sim_o u : \exists v' \in \Sigma^* : u'av' \in B. \quad (1)$$

The distributed *closure* of a behavior, denoted $\overline{B}$, adds just enough executions to make B distributed.

Definition 3 (Distributed Closure ($\overline{B}$)) *For every u, v, a, with a being controlled by object o,*

$$uav \in \overline{B} \iff \exists u' \sim_o u : \exists v' \in \Sigma^* : u'av' \in B. \quad (2)$$

Lemma 1 *Consider some behavior B. For every distributed behavior B', it is the case that*

$$B \subseteq B' \implies B \subseteq \overline{B} \subseteq B'.$$

Proof 1 *Clearly, $\overline{B} \supseteq B$, for $\sim_o$ is reflexive. Suppose that $B \subseteq B'$ but there is some execution in $\overline{B}$ which is not in B', i.e. $uav \in B$ and $uav \notin B'$. However, since $uav \in \overline{B}$, then there is some $u' \sim_o u$ and v' such that $u'av' \in B$. Therefore, $u'av' \in B'$, because $B \subseteq B'$. But this is not possible, as (i) B' is a distributed implementation and, (ii) by hypothesis, there is no continuation v'' such that $uav'' \in B'$. Therefore, there is no continuation v''' such that $uav''' \in B'$ (since $u \sim_o u'$). We reach a contradiction.*

It is also easy to check that $\overline{B}$ is distributed.

Under this constraint, it is easy to devise a QHMSC with no distributed implementation [2, 3]. Our goal is to allow all the executions modeled in the QHMSC and, yet, to add as few executions as possible:

$$[\![M]\!]p \subseteq B \wedge \nexists \text{ distr. } B' : [\![M]\!]p \subseteq B' \subset B. \quad (3)$$

Hence, we want to synthesize the "least imprecise" distributed implementation. The scenarios which are added to M, because of distribution, are called *implied scenarios* and there are algorithms to detect and report on their presence, in collections of MSCs [2] or HMSCs [23]. We plan to adapt these algorithms to detect implied scenarios in QHMSCs as well, but this is out of the scope of the present paper.

This criterion states that an object o performs an event a, after some execution u, if a is a valid continuation of some execution of the QHMSC (say, u') which looks exactly as u, from o's point of view.

We prove that the distributed closure of the QHMSC satisfies our optimality criterion.

Theorem 1 *B is an optimal distributed implementation of M (wrt (3)) iff $B = \overline{[\![M]\!]\ p}$.*

Proof 2 *Follows from Lemma 1 and the fact that $\overline{[\![M]\!]\ p}$ is distributed.*

3.2. Algorithm

The following conditions about the input QHMSC M are preconditions for our algorithm.

1. Within a class, renaming does not alter the semantics of the language. Intuitively, a single instance of a class may not qualify to play a particular role, which implies a distinct behavior, wrt other members of the class. Thus, letting M be a QHMSC, for every population p and every pair of objects i, j such that $p(i) = p(j)$,

$$[\![M]\!]p \quad = \quad ([\![M]\!]p)\,[i \leftrightarrow j], \tag{4}$$

   where $[i \leftrightarrow j]$ represents the substitution of all i's by j's and vice-versa. For short, a class may only contain a constant or variables, in the QHMSC.

2. Let M' be a universally quantified sub-term of M, i.e. $M' = \forall x : C\,(\phi \implies M'')$. Then, all messages appearing in M'' shall *depend* on x: x must be the sender, the receiver or a parameter of the message.

3. Let M' be a "parallel" sub-term of M, i.e. $M' = M_1 \parallel M_2$. Then, M_1 and M_2 alphabets are disjoint.

The first precondition will be relaxed by the end of Section 3.3. The second and third preconditions, which are strong, are needed to ensure that the semantics of the algebraic constructs of state machines match the semantics of QHMSC constructs.

We follow the usual scheme for synthesizing state machines from HMSCs [17], but we add a new step to deal with quantification. Our goal is to synthesize a state machine for a given class C, from a QHMSC. More precisely, we have to build a class-level state machine, whose instantiation, for every object having C as actual class, will fulfill the requirement of equation (2). To do so, our algorithm performs sequentially the following operations:

Instantiation of the QHMSC. It inserts an instance, named "self", wherever an instance with actual class C can appear in the QHMSC. The essential property preserved by this algorithm (presented in Table 1), is stated in Lemma 2.

Projection of a QHMSC onto instance "self". This removes all lifelines which do not belong to "self", from the QHMSC. This operation abstracts the QHMSC, keeping only the local view of instance "self".

Translation of the projected QHMSC to a state machine, mapping every construct (sequence, to its corresponding constructor in state machines. This step is simple, provided there are enough constructs in the state machine description [29]. As we added quantification in *both* notations, we avoid the difficulties of previous synthesis algorithms [28].

Post-processing the state machine. This includes interactive or automated prettification, to improve its readability. We do not describe this step here, for lack of space, but the reader is referred to [17, 18, 19]. It can also include some automatic refinement, to remove quantification and turn it into loops, for example. This latter step is needed if one wants to reuse code generation software that do not support quantifiers.

Our instantiation algorithm does not alter the semantics of the QHMSC, for non-empty classes. It inserts a new instance "*self*", which is *never quantified*.

Lemma 2 *For every population p, such that $p(\mathit{self}) = C$,*

$$[\![\mathit{inst}(C, M)|_{\mathit{self}}]\!]p \quad = \quad \big([\![M]\!]p\big)|_{\mathit{self}}$$

Thus, *self* can stand for any instance of C (following assumption (4)), in any population, without changing the semantics of the diagram. The QHMSC obtained before is projected. That is, the next phase of the algorithm proceeds with $\mathit{inst}(M)|_{\mathit{self}}$.

Building a state machine from a QHMSC is inspired from the construction of automata from regular expressions. The reader is referred to standard textbooks for these transformations, e.g. [15]. The function *sm*, which synthesizes this state machine is defined inductively, on the structure of the QHMSC, in Fig. 3.

$B \in$ **MSC:** Since B is finite, it has a finite number of executions. Those can easily be recognized by a state machine, which simply enumerates all of them. In full detail: the projection of B on *self* yields a Labeled Partial Order (LPO) on Σ, i.e. a tuple $\langle L, \prec, \lambda \rangle$, with L a finite set of *locations*, $\prec \subseteq L \times L$ a partial order on L and $\lambda : L \to \Sigma(\mathit{self})$ a labeling function, assigning to every location an event performed by *self*. Letting $L = \{l_1, \ldots, l_n\}$, a linearization of an LPO is a word $\lambda(l_1) \ldots \lambda(l_n)$ such that $l_j \prec l_i \implies j < i$. From every finite LPO, one can derive a state machine generating exactly its set of linearizations. Its set of states are "cuts" in the lpo, i.e. downwards-closed sets of locations. Formally, c is a cut if $\forall l, l' : (l \in c \wedge l' \prec$

$$
\begin{array}{rcl}
\mathit{inst}(C, B) & = & B \quad \text{where } B \in \mathsf{MSC} \\
\mathit{inst}(C, M_1 \parallel M_2) & = & \mathit{inst}(C, M_1) \parallel \mathit{inst}(C, M_2) \\
\mathit{inst}(C, M_1 \,;\, M_2) & = & \mathit{inst}(C, M_1) \,;\, \mathit{inst}(C, M_2) \\
\mathit{inst}(C, M_1 + M_2) & = & \mathit{inst}(C, M_1) + \mathit{inst}(C, M_2) \\
\mathit{inst}(C, M^*) & = & \mathit{inst}(C, M)^*
\end{array}
$$

$$
\mathit{inst}(C, \forall x : C' \, (R(t,x) \implies M)) = \begin{cases} \text{if } C \sqsubseteq C' & R(t, \mathit{self}) \implies \mathit{inst}(C, M)[x/\mathit{self}] \\ & \parallel \quad \forall x : C' \, ((R(t,x) \wedge x \neq \mathit{self}) \implies \mathit{inst}(C, M)) \\ \text{else} & \forall x : C' \, (\mathit{inst}(C, M)) \end{cases}
$$

$$
\mathit{inst}(C, \exists x : C' \, (R(t,x) \wedge M)) = \begin{cases} \text{if } C \sqsubseteq C' & R(t, \mathit{self}) \wedge \mathit{inst}(C, M)[x/\mathit{self}] \\ & + \quad \exists x : C' \, ((R(t,x) \wedge x \neq \mathit{self}) \wedge \mathit{inst}(C, M)) \\ \text{else} & \exists x : C' \, (\mathit{inst}(C, M)) \end{cases}
$$

Table 1. Instantiation algorithm

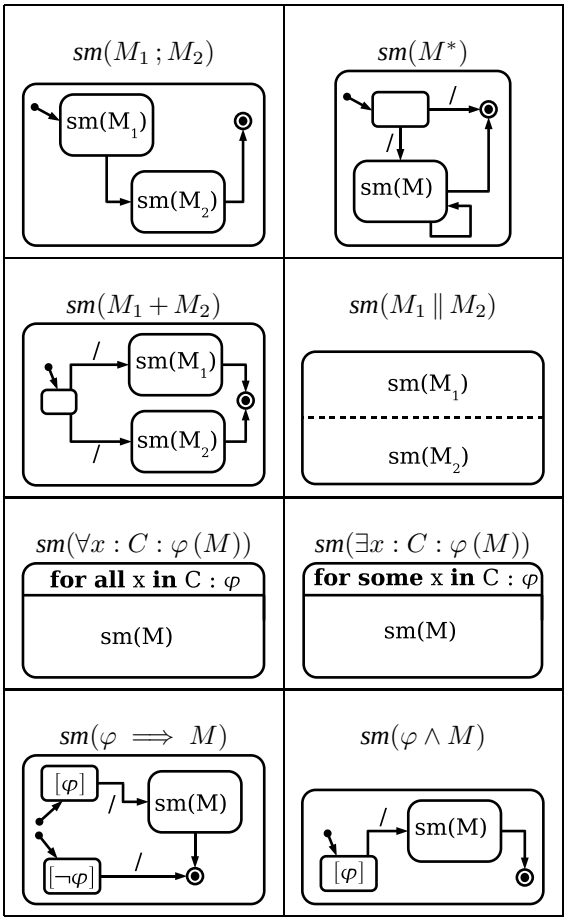

Figure 3. Definition of *sm* function

$l) \implies l' \in c$. There is a transition from some cut c to c', labeled by $e \in \Sigma$ if there is a location l such that

1. $c' = c \cup \{l\}$,
2. $l \notin c$,
3. $\forall l' \prec l : l' \in c$,
4. $\lambda(l) = e$.

The initial state is $\emptyset$ and the final state is L. This state machine recognizes $[\![B]\!]$.

$M = M_1 \,;\, M_2$**:** $\mathit{sm}(M_2)$ is launched as soon as $\mathit{sm}(M_1)$ completes, thanks to a completion transition.

$\mathit{sm}(M^*)$**:** Either $\mathit{sm}(M)$ is not entered (0 iterations) or it is entered, and upon exit, the machine non-deterministically chooses to launch $\mathit{sm}(M)$ again or to terminate.

$M = M_1 \parallel M_2$**:** In this case, $\mathit{sm}(M_1)$ and $\mathit{sm}(M_2)$ yield two state machines, S_1 and S_2. A new state machine is built, consisting solely of an orthogonal region. This region contains S_1 and S_2. Thanks to the preconditions stated above, we have $[\![\mathit{sm}(M_1 \parallel M_2)]\!] \; p = [\![M_1 \parallel M_2]\!] \; p$.

$M = M_1 + M_2$**:** The new state machine chooses non-deterministically between $\mathit{sm}(M_1)$ and $\mathit{sm}(M_2)$. This corresponds to ϵ-transitions in classical automata theory. As our semantics is linear, it is the case that $[\![\mathit{sm}(M_1 + M_2)]\!] \; p = [\![M_1 + M_2]\!] \; p$.

$M = \forall x : C : \varphi \, (M)$**:** Simply universally quantify $\mathit{sm}(M)$, with the expression for all $x : C : \varphi$. Again, semantics is preserved, thanks to the preconditions.

$M = \exists x : C\,(M)$**:** Existentially quantify $sm(M)$ with expression for some $x : C : \psi$. Semantics is preserved, too.

$\varphi \implies M$**:** means "if condition φ is true, behave as M, otherwise do nothing". Its translation relies on invariants, see Section 2.2.

$\varphi \wedge M$**:** asserts that φ holds before starting M. They are akin to hot/cold locations in LSCs.

By construction and thanks to our preconditions, our translation of QHMSCs into state machines preserve the semantics of the former.

Lemma 3 (*sm* preserves the semantics)

$$[\![sm(M)]\!]\, p \quad = [\![M]\!]\, p.$$

As informally described above, our synthesis algorithm is

Definition 4 (Synthesis Algorithm)

$$sm(inst(C, M)|_{self}).$$

By construction, the resulting state machines is the best implementation of the QHMSC.

Theorem 2 *For every class C, $[\![sm(inst(C, M)|_{self})]\!]p$ fulfills criterion* (2)*.*

Proof 3 *Combining Lemmata 2 and 3, we obtain that*

$$\begin{aligned}[\![sm(inst(C, M)|_{self})]\!]\, p &= [\![inst(C, M)|_{self}]\!]\, p \\ &= \big([\![M]\!]\, p\big)|_{self},\end{aligned}$$

which immediately implies that $[\![sm(inst(C, M)|_{self})]\!]\, p$ fulfills criterion (2)*.*

3.3. Example

The "Simple Update System" (SUS) is a very simple object-oriented system, the structure of which is presented in Fig. 4. It is made of a central server, to which clients connect. Some operators are authorized on this server; they may manually trigger update. When it occurs, the server must send a query to *all* its connected clients, asking them to fetch some data from their data sources (scenario "update", see Fig. 7). The clients pick some of their download mirrors and start downloading this new data. If download fails, they notify the server, which will, after every client has answered, disconnect all clients. If all clients manage to get the new data, everything is fine. The whole example is presented in the technical report version of this paper [8]. Although it is a toy example, it features the main particularities from the TRACON system of CTAS [28, 7] which led us to extend the state machines and scenario notations.

The two main scenarios, viz. "connection" and "update", are presented in Fig. 5 and 7. Those scenarios refer to another scenario, named "Disconnect All", which is presented in Fig. 8. Our synthesis algorithm has been applied to "connection" for class `Client` (see Fig. 9) and to the "update" scenario for `Server` (see Fig. 10).

Finally, we discuss the assumption of Eq. (4). Suppose that some instance is referred to by its particular name in the QHMSC, for example, one of the "clients" is special, in the sense that it can shutdown the server. This client is known by its name ("master"). Then, one can ask to synthesize a machine *specially for master*. The instantiation algorithm simply replaces all occurrences of "master" by "*self*", in addition to instantiating the class "client". Thus, eq. (4) is not really needed for the proper working of our algorithm. We introduced it for facilitating the presentation. In summary, we introduce a subclass for each constant.

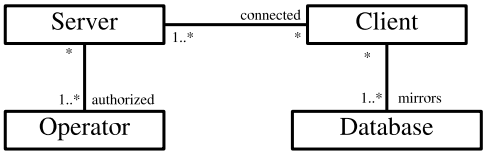

Figure 4. Class Diagram for "SUS"

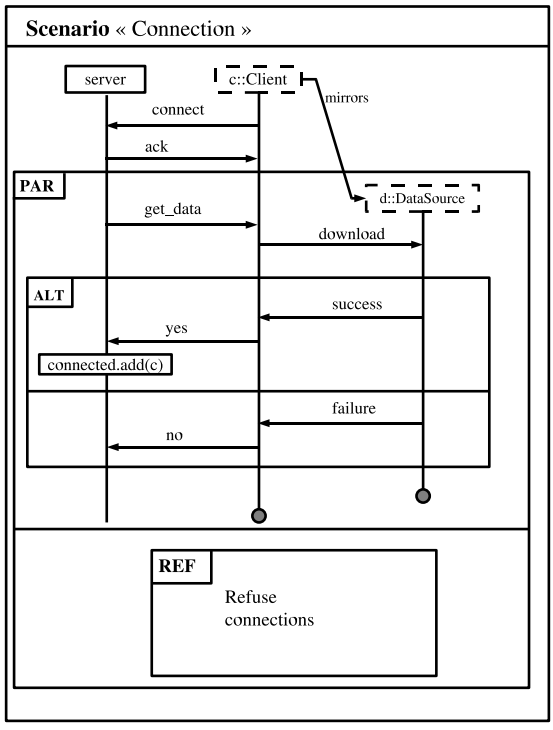

Figure 5. Connection Scenario

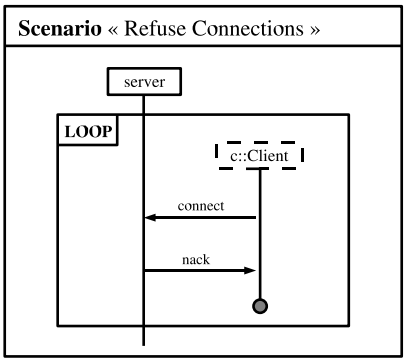

Figure 6. Refuse Connections Scenario

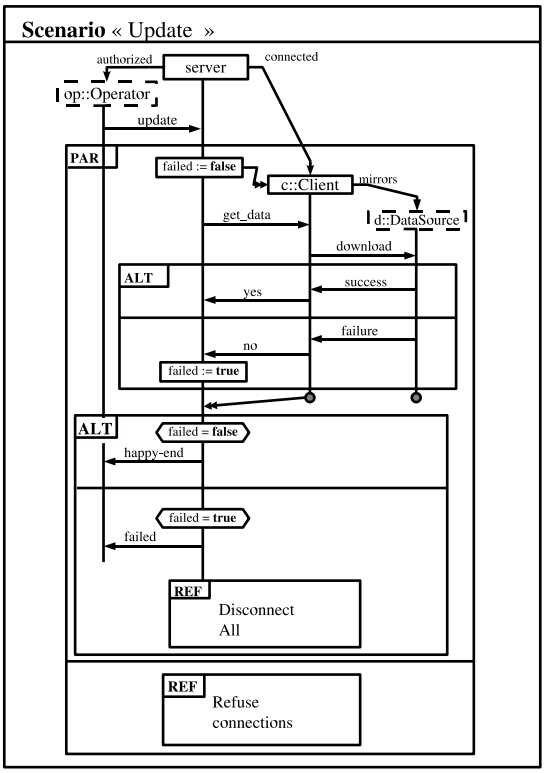

Figure 7. Update Scenario

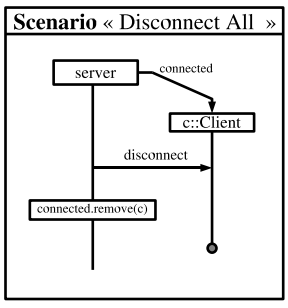

Figure 8. Disconnection Scenario

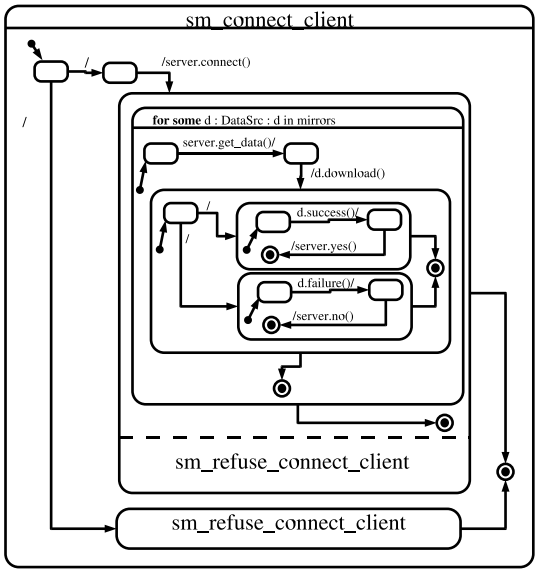

Figure 9. State Machine for Client (Connect Scenario)

4. Related Work

Many synthesis algorithms, building state machines from scenarios, have been developed and we will only cite some of them, stressing the different approaches.

The problem which mainly fostered researcher's attention is combining automatically various partial scenarios into a whole specification. For instance, Koskimies *et al.* present in [16], an algorithm based on Biermann-Krishnaswamy's method [4]. The underlying idea is that scenarios are examples of execution, and that actions are merged, with the idea that they should have the same effects. In [17], MSCs are labeled with global state descriptions and glued together accordingly. In [27], messages are given pre/post-conditions to infer some information about the global state. Otherwise, the two latter algorithms, as well as [29], follow essentially the same approach as ours to construct a state machine. Other techniques, in the spirit of Discrete Event System controller synthesis are followed by Kugler and Harel [13], and by Bontemps *et. al.* [9]. However, they base their work on Live Sequence Charts (LSCs), which are more expressive than MSCs, but also more complex to automatically analyze.

Nevertheless, all these approaches remain at instance-level. The need for class-level scenario-based descriptions was first spotted by Marelly *et al.* [20], who proposed an extension of LSCs to deal with this, together with an adapted play-out algorithm. This work was a great source of inspiration for our own notation. In [7], this notation has been successfully applied to the modeling of the TRACON part

of CTAS. On the very same case study, Whittle *et. al.* [27] report on the successful application of their algorithm, but they also acknowledge the need for notation and algorithmic extensions capable of coping, in full generality, with class-level scenarios. This is exactly what we do. Hence, this paper fills an important gap in the landscape of behavioral modeling.

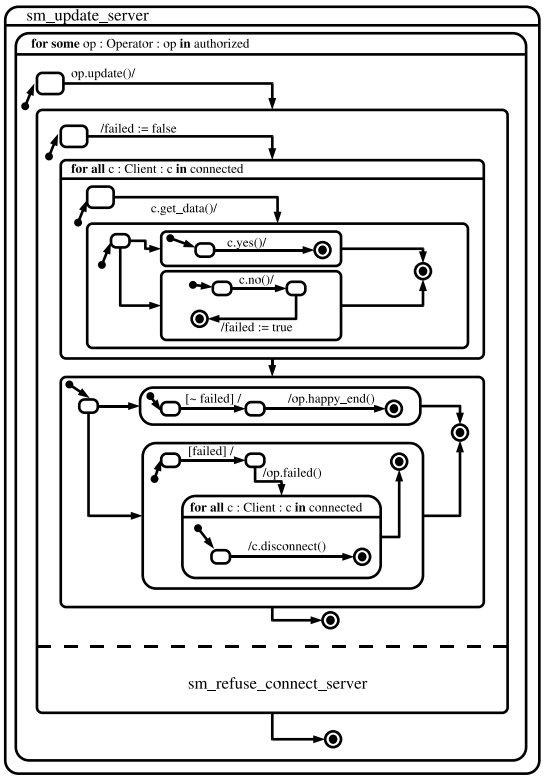

Figure 10. State Machine for Server (Update Scenario)

5. Conclusion and Future Work

We have presented a straightforward extension to behavioral notations for object-oriented systems, in order to enable "class-level descriptions" of such system. This extension adds universal and existential quantification over instances to the usual notations of (hierarchical/concurrent) state machines (Statecharts) and high-level scenarios (HMSCs). Classical synthesis algorithms, building a set of state machines from a structured scenario-based model, have been accordingly modified, in order to work at class-level. We have illustrated these algorithms on a small example. Our notations are fully formal, equipped with a compositional semantics (in Plotkin's Structured Operational Semantics style), which we did not present here, for the sake of readability.

We opened much more questions than we answered. How can one detect and report on implied scenarios in QHMSC? What methodology must be followed to take a couple of instance-level, unrelated, scenarios and derive from them the integrated class-level scenario needed by our algorithm? We need to generate code from our models, so that it can really be useful in model-driven approaches (MDA) or in rapid prototyping. In this case, how can one bridge the gap between the generated program and an existing implementation, a problem already highlighted by [28]? Is it possible to integrate other useful constructs, such as variability (for software product lines), real-time, preemption? Can other notations, such as Activity Diagrams, be made "class-level" as well?

References

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comp. Sci.*, 126:183–235, 1994.

[2] R. Alur, K. Etessami, and M. Yannakakis. Inference of Message Sequence Charts. In *Proceedings of 22nd International Conference on Software Engineering*, pages 304–313, 2000.

[3] H. Ben-Abdallah and S. Leue. Syntactic Detection of Process Divergence and Nonlocal Choice in Message Sequence Charts. In E. Brinksma, editor, *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, Third International Workshop, TACAS'97*, number 1217 in LNCS, pages 259–274, Enschede, The Netherlands, 1997. Springer-Verlag.

[4] A. W. Biermann and R. Krishnaswamy. Constructing Programs from Example Computations. *IEEE Transactions on Software Engineering (TSE)*, SE-2(3):141–153, September 1976.

[5] Y. Bontemps and P. Heymans. Turning High-Level Live Sequence Charts into automata. In *Proc. of "Scenarios and State-Machines: models, algorithms and tools" (SCESM) workshop of the 24th Int. Conf. on Software Engineering (ICSE 2002)*, Orlando, FL, May 2002. ACM. `http://www.cs.tut.fi/~tsysta/ICSE/papers/`.

[6] Y. Bontemps and P. Heymans. As fast as sound (lightweight formal scenario synthesis and verification). In H. Giese and I. Krüger, editors, *Proc. of the 3rd Int. Workshop on "Scenarios and State Machines: Models, Algorithms and Tools" (SCESM'04)*, pages 27–34, Edinburgh, May 2004. IEE. available at `http://www.info.fundp.ac.be/~ybo`.

[7] Y. Bontemps, P. Heymans, and H. Kugler. Applying LSCs to the specification of an air traffic control system. In S. Uchitel and F. Bordeleau, editors, *Proc. of the 2nd Int. Workshop on "Scenarios and State Machines: Models, Algorithms and Tools" (SCESM'03), at the 25th Int. Conf. on Soft. Eng. (ICSE'03)*, Portland, OR, USA, May 2003. IEEE. available at `http://www.info.fundp.ac.be/~ybo`.

[8] Y. Bontemps, P. Heymans, G. Saval, P.-Y. Schobbens, and J.-C. Trigaux. Class-level behavioral modeling and syn-

thesis. Technical report, University of Namur, Institut d'Informatique, August 2004.

[9] Y. Bontemps, P.-Y. Schobbens, and C. Löding. Synthesizing open reactive systems from scenario-based specifications. *Fundamenta Informaticae*, XX:1–31, 2004.

[10] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[11] D. Harel. Statecharts: A visual formalism for complex systems. Technical report, Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel, July 1986. February 1984, extensively revised February 1986.

[12] D. Harel. From play-in scenarios to code : An achievable dream. *IEEE Computer*, 34(1):53–60, January 2001. a previous version appeared in Proc. of FASE'00, LNCS(1783), Springer-Verlag.

[13] D. Harel and H. Kugler. Synthesizing State-Based Object Systems from LSC Specifications. *International Journal of Foundations of Computer Science*, 13(1):5–51, February 2002. (Preliminary version appeared in, *Proc. Fifth Int. Conf. on Implementation and Application of Automata (CIAA 2000)*, LNCS 2088, July 2000.).

[14] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: the STATEMATE Approach*. McGraw-Hill, 1998. ISBN 0-070-26205-5.

[15] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2nd edition, 2001.

[16] K. Koskimies, T. Männistö, T. Systä, and J. Tuomi. SCED: A Tool for Dynamic Modelling of Object Systems. Technical Report Report A-1996-4, Department of Computer Science, University of Tampere, University of Tampere, Department of Computer Science, P.O. Box 607, FIN-33101 Tampere, Finland, July 1996. ISBN 951-44-4003-X, ISSN 0783-6910.

[17] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to Statecharts. Kluwer Academic Publishers, 1999. Franz J. Rammig (ed.).

[18] I. H. Krüger. *Distributed System Design with Message Sequence Charts*. PhD thesis, Technischen Universität München, July 2000.

[19] S. Leue, L. Mehrmann, and M. Rezai. Synthesizing ROOM models from message sequence charts specifications. In *Proc. of 13th IEEE Conference on Automated Software Engineering*, Honolulu, Hawaii, October 1998.

[20] R. Marelly, D. Harel, and H. Kugler. Multiple Instances and Symbolic Variables in Executable Sequence Charts. In *Proc. 17th Ann. ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'02)*, pages 83–100, Seattle, WA, 2002.

[21] J. Mukerji and J. Miller. Mda guide v 1.0.1 (omg), March 2003.

[22] Object Management Group (UML Revision Task Force). *OMG UML Specification (2.0)*, September 2003. `http://www.omg.org/uml`.

[23] S. Uchitel. *Elaboration of Behaviour Models and Scenario-based Specifications using Implied Scenarios*. PhD thesis, Imperial College London, January 2003.

[24] S. Uchitel and J. Kramer. A Workbench for Synthesizing Behaviour Models from Scenarios. In *Proc. of the 23rd IEEE International Conference on Software Engineering (ICSE'01)*. ACM, 2001.

[25] I. T. Union. MSC-2000: ITU-T Recommendation Z.120 : Message Sequence Chart (MSC), 2000. `http://www.itu.int/`.

[26] M. von der Beeck. A structured operational semantics for UML-statecharts. *Jour. on Software and Systems Modeling*, 1(2):130–141, December 2002.

[27] J. Whittle and J. Schumann. Generating Statechart Designs from Scenarios. In *22nd International Conference on Software Engineering (ICSE 2000)*, pages 314–323, Limerick, Ireland, June 2000. ACM.

[28] J. Whittle and J. Schumann. Statechart Synthesis from Scenarios: an Air Traffic Control Case Study. In *Proc. of "Scenarios and State-Machines: models, algorithms and tools" workshop at the 24th Int. Conf. on Software Engineering (ICSE 2002)*, Orlando, FL, May, 20th 2002. ACM. `http://www.cs.tut.fi/~tsysta/ICSE/papers/`.

[29] T. Ziadi, L. Hélouët, and J.-M. Jézéquel. Revisiting statechart synthesis with an algebraic approach. In *Proc. of 26th International Conference on Software Engineering (ICSE)*, pages 242–251, Edinburgh, May 2004. IEEE Computer Society. ISBN 0-7695-2163-0.