

Automated Software Engineering Using Concurrent Class Machines

Radu Grosu Yanhong A. Liu Scott Smolka Scott D. Stoller Jingyu Yan

State University of New York at Stony Brook
Stony Brook, NY 11794-4400, USA
{grosu,liu,sas,stoller,jingyu}@cs.sunysb.edu

Abstract

Concurrent Class Machines are a novel state-machine model that directly captures a variety of object-oriented concepts, including classes and inheritance, objects and object creation, methods, method invocation and exceptions, multithreading and abstract collection types. The model can be understood as a precise definition of UML activity diagrams which, at the same time, offers an executable, object-oriented alternative to event-based statecharts. It can also be understood as a visual, combined control and data flow model for multithreaded object-oriented programs. We first introduce a visual notation and tool for Concurrent Class Machines and discuss their benefits in enhancing system design. We then equip this notation with a precise semantics that allows us to define refinement and modular refinement rules. Finally, we summarize our work on generation of optimized code, implementation and experiments, and compare with related work.

1. Introduction

Embedded software design automation (ESDA) is emerging as a promising approach to developing the high-confidence software demanded by embedded-system applications, including those found in the telecommunications, aerospace/military, medical-device, and automotive industries. A comprehensive ESDA solution is expected to provide integrated support for (1) requirements capture, (2) visual modeling, (3) simulation, (4) component mapping to target platforms, (5) code generation, (6) testing and (7) documentation generation.

Visual modeling plays a key role in this context, akin to the “design blueprints” found in other engineering disciplines such as electrical, mechanical and civil. Errors undetected at this stage of development are carried over to later stages where they are much more costly to deal with. The importance of visual modeling has been recognized by the Object Management Group (OMG) computer-industry consortium. The goal of this group is to set software standards that enable distributed and enterprise-wide interoperability.

OMG is arguably most well known for UML, the unified modeling language that has been unanimously embraced by industry and promises to become the visual modeling language of ESDA.

To model the behavior of object-oriented systems, UML proposes both a state model (given by state diagrams) and a flow model (given by activity diagrams). The former emphasizes states and state hierarchy, while the latter emphasizes actions and action hierarchy. Emphasizing states leads to an intuitive understanding of allowed method-invocation sequences. Emphasizing actions leads to an intuitive understanding of exceptions, recursion and inheritance. A recent RFP (Request For Proposals) for the UML 2.0 Superstructure asked for clarification of the precise relation between class and behavior diagrams. In particular, the RFP asks: How does message-based communication relate to method invocation and how is inheritance captured in behavior diagrams (state machines cannot be generalized in UML)?

In this paper, we develop a comprehensive framework for automated software engineering based on a combined control and data flow model for multi-threaded object-oriented programs we call *concurrent class machines* (CCMs). CCMs can be understood as a precise semantic definition of UML activity diagrams that offers an executable object-oriented alternative to event-based statecharts. It can also be understood as an abstract, visual model for Java programs. The visual presentation can enhance developer productivity and the interaction between developers and customers. It also enables the development of efficient analysis algorithms that exploit the structure present in the model. Our main contributions are the following.

- We introduce a visual notation for CCMs which we refer to as *Visual Class Machines* (Section 2). Visual CM provides an intuitively appealing notation for developers that graphically and coherently captures a variety of object-oriented features, including classes and inheritance, objects and object creation, methods, method invocation and exceptions, multi-threading and abstract collection types.
- We then present our *Concurrent Class Machines* model (Sections 3 and 4) which serves both as a for-

mal, transition system-style semantics and abstract syntax for Visual CM. The operational nature of CCMs renders Visual CM specifications *executable* and the model itself provides the basis for powerful analysis, verification, and code-generation techniques.

- We complement our CCM model with a collection of *refinement rules* (Section 5) that allow one to reason compositionally about multi-threaded, recursive, object-oriented systems in a trace-based setting. The basis for our definition of CCM traces is the notion of an evolving *communication interface* between a system and its environment.

2. A Visual Language for Class Machines

We first describe Visual CM, a visual representation for CMs. (We have also developed a grammar for a textual representation of CMs.) By design, the visual representation and CMs have exactly the same structure, so CMs provide an “abstract syntax” as well as a semantics for the visual language. Therefore, we do not give a separate grammar for the visual language. Instead, we present the language using a familiar example—the readers/writers problem. Next, a semantically minimal extension to handle concurrency is introduced, providing a notation for concurrent CMs.

2.1. Visual CM

A solution to the readers/writers problem—effectively, an implementation of read locks and write locks—appears in Figure 1. To save space, the figure does not show class `Resource`, which declares public methods `read():int` and `write(int x)`, and class `MonitorExc`, which declares no fields or methods; definitions of methods of `WrCap` are also elided. Our solution is fairly standard except for the use of capabilities to identify the client to whom a lock was granted; more commonly, thread identity is used. Our approach is more flexible. For example, a coordinator thread could create a resource, a monitor, and a write capability, acquire the write lock embodied in the write capability, initialize the resource, and then pass the write capability to a worker thread, which can immediately use the write capability to access the resource. The capabilities also enforce the calling discipline; for example, if the acquire operation is invoked on a capability whose lock is already held, the capability throws a `MonitorExc`. Our language is designed to fit into the UML, so class diagrams are used to describe the static structure of systems. Due to space limitations, we omit the class diagram for our example and include information from it directly in the UML-like activity diagrams that define the behavior of methods. The notation is the same as in class diagrams: underlining a declaration indicates that the declared element has class scope (this is

equivalent to Java’s `static` modifier), and the visibility modifiers are public (“+”), private (“-”), and package (default). We use a Java-like `throws` clause to indicate the types of exceptions that a method can throw.

Methods are defined by UML-like activity diagrams. Three kinds of nodes may appear on the border of a diagram: (1) a *call entry node* (unfilled circle),¹ (2) *return exit nodes* (filled circles), each labeled with an expression that evaluates to the return value, (3) *exception exit nodes* (filled diamonds), each labeled with an expression that evaluates to the returned exception.² The types of expressions associated with return nodes must be consistent with the declared return type of the method; similarly, the types of expressions associated with exception nodes must be consistent with the `throws` clause in the declaration of the method.

Execution of a CM starts by creating an instance of the designated main class and invoking its `run` method. Execution of a method starts at the call node and proceeds along edges labeled with *guarded commands*, which consist of a *guard* (a Boolean expression) and a *parallel assignment statement* (a set, possibly empty, of assignments that are performed in parallel). Naturally, an edge can be traversed only if the guard is true. We elide the constant guard `true` and the empty parallel assignment statement.

Guarded commands may contain only operations that have no net effect on the size of the stack or heap. Thus, expressions may contain operations on primitive values (integers, sets regarded as mathematical values, etc.), reads of object attributes, and invocations of functional methods (i.e., methods that do not throw exceptions and do not contain writes to object attributes); parallel assignments may update variables and object attributes.

Operations that may have a net effect on the size of the state appear in *boxes*. The sequential language contains two kinds of boxes: *method invocation boxes*, which represent method invocations with dynamic dispatch, and *object creation boxes*, which allocate and new object, execute a constructor to initialize it, and return a reference to it. Our language is garbage-collected and hence does not contain object de-allocation boxes. Each box has a call-entry node, labeled with the argument, a return-exit node, labeled with a variable or attribute into which the returned value is stored, and (if the called method can throw an exception) an exception-exit node, labeled like the return-exit node.

An *internal node* is a node that is not a call, return, or exception node of a method definition or box. Internal nodes are simply intermediate control points. Execution blocks when control is at an internal node whose outgoing edges all have false guards. This is a well-known, flexible, high-level

¹In the formal CM model, the call node is labeled with the formal parameters; in the visual language, the formal parameters appear in the method declaration.

²If the return or exception type is `void`, the corresponding expression may be omitted.

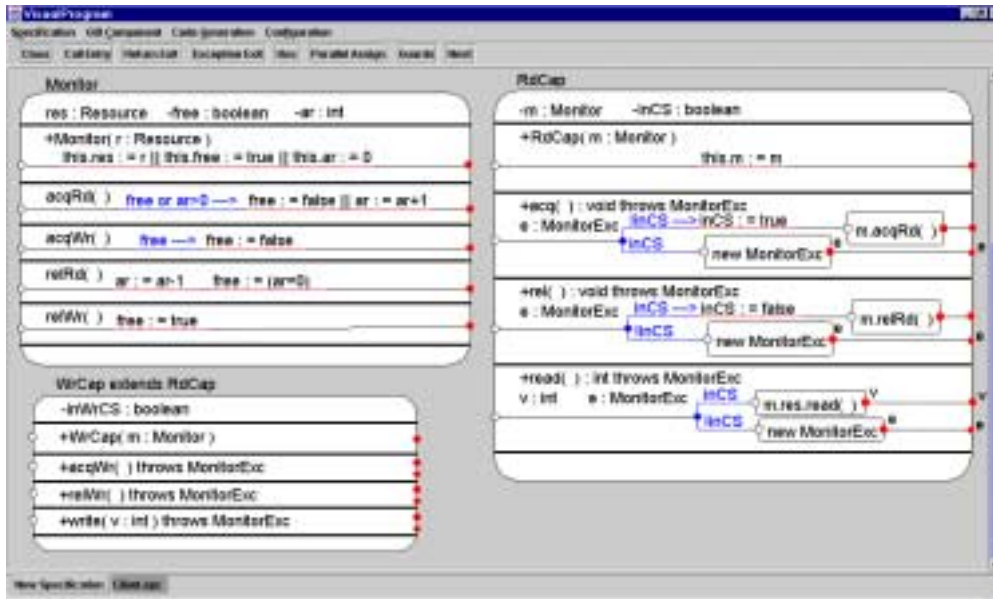


Figure 1. Package ReaderWriter.

way to express condition synchronization in concurrent systems. For example, execution of `Monitor.acqWr` blocks if the monitor is not free. Declarations of local variables may appear anywhere inside the method body (e.g., the declaration of `e` in `RdCap.acq`).

Conditional branches are drawn with unfilled diamonds; this is syntactic sugar, just as it is in activity diagrams [7, p. 263]. An example appears in the definition of `RdCap.acq`. As in activity diagrams, the guards on the outgoing transitions of a conditional diamond should be disjoint and exhaustive [7, p. 263].

For brevity, some features described in this section are omitted from the formal definition of CMs in Section 3 but can easily be added. Specifically, these features are: static attributes and methods, multiple exception types for a single method, packages, and visibility modifiers for methods and attributes. Also, the semantics for object creation reflects only the behavior default constructors, which have no arguments (except `this`) and initialize all attributes to default values (zero for integers, null for references, and so on); non-default constructors can easily be modeled as this default initialization followed by invocation of the specified constructor.

2.2. Visual Language for Concurrent CMs

Concurrent CMs are an extension of CMs with a mechanism for forking threads. Incorporating only this semantically minimal extension into the language helps simplify the semantics and allow maximum flexibility. Synchronization is achieved using guarded commands. The semantics requires: (1) a guarded command executes atomically, and (2)

it executes only in states where its guard is true. All common synchronization constructs (semaphores, locks, condition variables, wait-free data structures, etc.) can be implemented using these primitives. We allow methods to be declared `atomic`; this is syntactic sugar for acquiring and releasing a system-wide lock (thus, `atomic` is stronger than `synchronized` in Java). An optimizing code generator might allow more concurrency by using multiple locks, when this is behaviorally equivalent.

An alternative would be to prohibit the use of guards for condition synchronization (by allowing guards only in conditional diamonds, where the guards should be exhaustive) and introduce selected synchronization constructs as primitives. This approach would clutter the semantics, and the choice of primitives would inevitably be somewhat arbitrary and unsatisfactory for some applications. For example, if Java-like monitors are primitive, there is no way to associate multiple condition variables with a single lock (this is inconvenient when solving the readers/writers problem, because a natural approach is to have separate condition variables on which readers and writers wait, and to associate these two condition variables with a single lock).

An object is associated with each thread. These objects are instances of a distinguished class named `Thread` or a subclass thereof. Class `Thread` declares one method with signature `run(): void` and no attributes. These objects are created and initialized in the usual way, using object creation boxes, etc. The thread of execution is started using a new kind of box, called a *thread start box*. An example appears in Figure 2. The new thread executes the `run` method of the target object. The thread start operation is a box because it changes the structure of the state space,

by allocating a new call stack. Execution of a concurrent CM starts in the same way as execution of a CM; the only additional requirement is that the main class is a subclass of Thread.

The diagram for `Client.run` contains a simple example of catching an exception. The diagram for `Client.activateClient` illustrates our notation for propagating exceptions. Consider a method invocation box b in a method m . Suppose b has an exception exit node n_b . It is common for m not to catch the exception. We make this the default behavior using the following syntactic sugar: if n_b has no explicit outgoing edges, then implicitly there is an edge labeled with `true --> skip` and going from n_b to an exception exit node n_m of m with appropriate type, and the exception expression associated with n_m is the variable associated with n_b (if that variable is omitted, then implicitly a fresh local variable is used at n_m and the corresponding n_b 's).

Synchronization bars are used in UML activity diagrams to represent forks and joins in a structured way. Synchronization bars can easily be introduced in our language as syntactic sugar: forks correspond to boxes that create and start threads, and joins correspond to a simple condition synchronization that can be implemented in various ways.

3. Sequential Class Machines

In this and the following section we introduce a class machine model that closely corresponds to the sequential and concurrent parts of the visual language respectively. Keeping the model close to the syntax makes it accessible to engineers. The model sets a solid ground for understanding their visual specifications and for developing tools. It also helps us to develop efficient analysis algorithms that exploit the structure present in the model.

As in C++ or Java, the class machine model allows recursion. The meaning of recursion is usually explained by introducing a least fixpoint operator. In this paper we use a more operational approach. We give the meaning of recursion by defining a flat state machine, with a potentially unbounded number of states and transitions that, similarly to the fixpoint operator, computes the least solution of the recursive machine by unfolding the method invocations. Note however, that our analysis algorithms work directly on the finite-control recursive machine.

3.1. The Recursive Definition

Environments and actions. We assume, the set of *classes* C and their *subclass relation* \leq with single inheritance is given by the class diagrams in our visual notation. Given a set $X = \{x_1:T_1, \dots, x_n:T_n\}$ of typed variables, a *variable environment (frame)* σ over X is a partial function $[x_1 \mapsto a_1, \dots, x_k \mapsto a_k]$ where $k \leq n$ and for all i , $a_i \in U_i$

and $U_i \leq T_i$. We denote by $\sigma[y \mapsto a]$ the environment where the value of y is bound to a . The set of all variable environments over X is denoted by Σ_X . Attributes and attribute environments are treated similarly to variables and variable environments. The set of attributes of each class is given by the class diagram.

For $x \in X$, let $\llbracket x \rrbracket$ be the *evaluation function* of x over σ defined by $\llbracket x \rrbracket_\sigma = \sigma(x)$ ³. For an expression e over X , let $\llbracket e \rrbracket$ be the homomorphic extension of the evaluation function to expressions. For example, suppose $\sigma(x) = 1$ and $\sigma(y) = 2$. Then $\llbracket x + y \rrbracket_\sigma = \llbracket x \rrbracket_\sigma + \llbracket y \rrbracket_\sigma = 1 + 2 = 3$. In the following, we call $\llbracket e \rrbracket$ also an expression and write it as e . Expressions are defined as usual over typed variables and identifiers of primitive (mathematical) functions.

To keep track of *data*, we maintain a frame σ and a global *object environment (object pool)* ω . Assuming that A is the set of all attributes and O is the set of all object identifiers, ω is a partial function in $C \rightarrow (O \rightarrow \Sigma_A)$. The domain of $\omega(c) : O \rightarrow \Sigma_{A_c}$ contains all instances of c . Define a partial function $\text{classOf}(o) = c$ iff $o \in \text{dom}(\omega_c)$. The set of all object pools ω is denoted by Ω . To keep track of *control*, i.e., the return locations of method calls and the current program counter, we maintain a *location stack* of boxes having on top the current node. Similarly, the variable frame is extended to a *stack of frames*.

An *action* from U to V is a relation $\alpha \subseteq U \times V$. Syntactically actions are *boxes* or *guarded commands*. Boxes encapsulate actions that change the size of environments. Guarded commands gc correspond to actions that test or change values in the data environments. They have the form $g \rightarrow a$ where the *guard* g is a boolean expression and the *parallel assignment* a is a set $x_1 := e_1 \parallel \dots \parallel x_n := e_n$ of assignments. We extend the evaluation function to guarded commands by defining $\llbracket gc \rrbracket_{\sigma, \sigma'}$ by its characteristic predicate $\llbracket g \rrbracket_\sigma \wedge \sigma' = \sigma[x_1 \mapsto \llbracket e_1 \rrbracket_\sigma, \dots, x_n \mapsto \llbracket e_n \rrbracket_\sigma]$. By definition, $\llbracket gc \rrbracket$ is an action from Σ_X to Σ_X .

To access object attributes, we extend expressions with *attribute selectors*. As a consequence, we extend the expression evaluation function to $\llbracket e \rrbracket_{\sigma, \omega}$ ⁴. The evaluation function for guarded commands is extended accordingly. For example, `!inCS -> inCS := true in RdCap.acq()` has for an object $v \in \text{dom}(\omega_{\text{RdCap}})$ the action:

$$\{(\sigma : \omega, \sigma' : \omega') \mid \neg(\omega_{\text{RdCap}}(v)(\text{RdCap.inCS})) \wedge \omega' = \omega[\text{RdCap} \mapsto \omega_{\text{RdCap}}[v \mapsto [\text{RdCap.inCS} \mapsto \text{true}]]]\}$$

Expressions *may not* contain method invocations, object creation or thread starting expressions. These operations appear only in boxes.

Definition of SCM. A *sequential class machine* $M = ((C, \leq), B, N, C_i, \Omega)$ consists of:

³For a function f we also write f_x for $f(x)$.

⁴To avoid parenthesis we write tuples (x, y) as $x:y$.

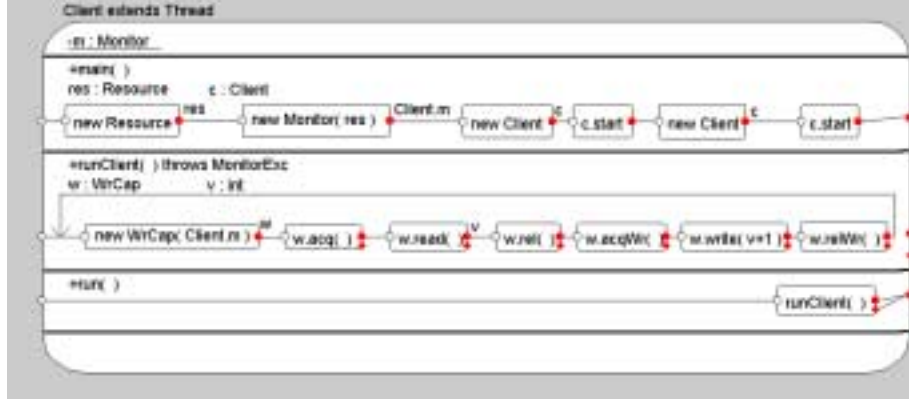


Figure 2. Package Client.

- A finite tree of classes (C, \leq) , where $c \leq d$ if class c is a subclass of class d . C_i is the main class. The machine starts by executing C_i 's run method.
- A finite set of boxes B , a finite set of nodes N and a set of all object environments Ω .

We assume that C, B and N are disjoint. Each class $c \in C$ has associated the following elements:

- A finite set of attributes $A_c = \{a_1:T_1, \dots, a_n:T_n\}$. The set of all attributes $\hat{A}_c = \cup_{d \geq c} \{d\} \times A_d$ consists of declared and inherited attributes prefixed with their class.
- A finite set M_c of methods.

Each method $m \in M_c$ has associated the following elements:

- A type $T_{cm} = cT_{cm} \times rT_{cm} \times eT_{cm}$ defining the call, return and exception types of m ⁵.
- A set of local variables $X_{cm} = \{x_1 : T_1, \dots, x_l : T_l\}$.
- A set of nodes $N_{cm} \subseteq N$ such that $N_{cm} \cap N_{dn} = \emptyset$ for $cm \neq dn$. This set is partitioned into:
 - A call entry node c with formal parameters $x : cT_{cm}$ and `this` : c in X_{cm} .
 - A set of return exit nodes R where each node $r \in R$ has a return expression re over X_{cm} of a type T where $T \leq rT_{cm}$.
 - A set of exception exit nodes E where each node $e \in E$ has an exception expression ee over X_{cm} of a type T where $T \leq eT_{cm}$.
 - A set of internal nodes I .
 - A set of box entry nodes $B_{cm}.En$ and a set of box exit nodes $B_{cm}.Ex$.

The set of from-nodes is defined as $F = \{c\} \cup I \cup B_{cm}.Ex$. The set of to-nodes is defined as $T = \{c\} \cup I \cup R \cup E \cup B_{cm}.En$.

⁵To simplify the presentation, we consider only one parameter and exception type.

- A set of invocation boxes $B_{cm} \subseteq B$ such that $B_{cm} \cap B_{dn} = \emptyset$ for $cm \neq dn$. The invocation boxes are partitioned into method invocation boxes and object creation boxes. Each method invocation box $b \in B_{cm}$ has associated:
 - An expression oe_b of a class type $d_b \in C$ determining the target object.
 - A call node $c_b \in B_{cm}.En$, a method identifier m_b with $m_b \in M_{d_b}$ and argument ae_b of a type T such that $T \leq cT_{d_b m_b}$.
 - A return node $r_b \in B_{cm}.Ex$ and a variable $x \in X_{cm}$ of a type T such that $T \geq rT_{d_b m_b}$ ⁶. The return value gets bound to x .
 - An exception node $e_b \in B_{cm}.Ex$ and a variable $y \in X_{cm}$ of a type T such that $T \geq eT_{d_b m_b}$. The exception value gets bound to y .

Object creation boxes are similar to method invocation boxes except that: (1) they have associated a class d_b instead of an object expression oe_b , (2) the call node has no expression, (3) the variable x of the return node has class d_b and (4) the method identifier is `new` and is not required to be in M_{d_b}

- A labeled transition relation $\delta_{cm} \subseteq F \times Act \times T$ where $Act = \mathcal{P}((\Sigma_{X_{cm}} \times \Omega) \times (\Sigma_{X_{cm}} \times \Omega))$ is a set of actions α from $\Sigma_{X_{cm}} \times \Omega$ to $\Sigma_{X_{cm}} \times \Omega$ that relate frame and pool tuples $\sigma_{cm} : \omega$.

For example, class `Client` has an attribute `Client.m` and the methods `main`, `runClient` and `run`. Method `runClient` starts with an object creation box having associated the class `WRCap`. The method invocation box `w.read()` has associated the expression (target object) `w`, the method identifier `read`, a return node with variable `v` and an exception node with no variable.

The above definition separates control from data. While control is kept explicitly in the nodes, data (environments)

⁶For primitive type T we assume \leq relates T to itself only.

is handled implicitly by the transitions. As a consequence the class machine has a *direct, compact* and *human readable* representation in visual CM. This representation also admits a compact symbolic representation (with BDDs) that is well suited for combined enumerative and symbolic model checking algorithms. By contrast, a state machine representation that adds the environments to the nodes, leads to a human unreadable representation and can easily blow up the state space in a symbolic representation.

3.2. Meaning of Recursion and Object Creation

The meaning of recursion and object creation in a CM $A = ((C, \leq), B, N, C_i, \Omega)$ is given in terms of a global class machine (GCM) $A^* = (GL, GF, n_i, \delta)$ consisting of the following elements:

- A set of *global locations* $GL \subseteq B^*N$. Each global location $\lambda = \beta:n$ of GL is a stack $\beta = b_1:\dots:b_k$ of boxes b_i with a node n on top. For each k, b_{k+1} (or n) belongs to a method m of class d where $d \leq c$, c is the class of the target expression oe associated with b_k and m is the method associated with b_k .
- A set of *global frames* $GF \subseteq \Sigma_X^*$ where X is the set of all variables. A global frame $\varphi \in GF$ is a stack $\sigma_0:\dots:\sigma_k$ associated with a location $b_1:\dots:b_k:n$. For each i , σ_i is the frame (local variable environment) of the method containing box b_{i+1} (or n). Let GF_λ be the set of all global frames associated with λ .
- A *labeled transition relation* $\delta \subseteq GL \times GA \times GL$ where $GA = \mathcal{P}((GF \times \Omega) \times (GF \times \Omega))$. This relation contains tuples $(\lambda_1, \alpha, \lambda_2)$ where λ_1, λ_2 are global locations and α is a global action from $GF_{\lambda_1} \times \Omega$ to $GF_{\lambda_2} \times \Omega$.

The transition relation δ is the least relation that contains for each method, internal transitions, call transitions, return transitions, exception transitions and object creation transitions. Call transitions capture dynamic method dispatch. To ease notation we treat environment tuples in $\Sigma_X \times \Omega$ and $GF \times \Omega$ also as stacks $\sigma : \omega$ and $\varphi : \omega$ respectively.

Internal. In this case the source location is $\lambda = \beta:n$ where n is a *from node* of a method $c.m$. If (n, α_{cm}, n') is in δ_{cm} and $\lambda' = \beta:n'$ then $(\lambda, \alpha, \lambda')$ is in δ where the global action α extends the action α_{cm} as defined below:

$$\{ (\varphi:\sigma:\omega, \varphi:\sigma':\omega') \mid (\sigma:\omega, \sigma':\omega') \in \alpha_{cm} \wedge \varphi:\sigma:\omega \in GF_\lambda \}$$

Call. In this case the source location is $\lambda = \beta:c_{b_k}$ where c_{b_k} is the call node of method box b_k . Let m, oe and d be the method, object expression and class of oe respectively, associated with box b_k . Let p be the class from which d inherits m ; if d defines m then $p=d$. For each class $a \leq p$ that defines m there is a destination location $\lambda' = \beta:b_k:c$ and a transition $(\lambda, \alpha, \lambda')$ in δ . The node c

is the call node of the method m of class a , and the global action α defined by

$$\{ (\varphi:\sigma:\omega, \varphi:\sigma:\tau:\omega) \mid \text{classOf}(oe_{\sigma,\omega}) = a \wedge \varphi:\sigma:\omega \in GF_\lambda \wedge \tau = \tau_i[\text{this} \mapsto oe_{\sigma,\omega}, x \mapsto ae_{\sigma,\omega}] \}$$

does the following: (1) it checks if the value of oe has class a ⁷, (2) it pushes a frame τ_i that binds local variables to default values and binds `this` and the formal parameter x to the values of oe and ae , respectively.

Return. In this case the destination location is $\lambda' = \beta:r_{b_k}$ where r_{b_k} is the return node of method box b_k . Let m, oe and d be the method, object expression and class of oe respectively, associated with box b_k . Let p be the class from which d inherits m ; if d defines m then $p=d$. For each class $a \leq p$ that defines m and each return node $r \in R_{am}$ there is a source location $\lambda = \beta:b_k:r$ and a transition $(\lambda, \alpha, \lambda')$ in δ , where the global action α defined by

$$\{ (\varphi:\sigma:\tau:\omega, \varphi:\sigma':\omega) \mid \sigma' = \sigma[x \mapsto re_{\tau,\omega}] \wedge \varphi:\sigma:\tau:\omega \in GF_{\lambda'} \}$$

does the following: (1) it binds the return variable x associated with node r_{b_k} of b_k to the actual return value of re in σ' and (2) it discards τ .

Exception. These transitions are handled similarly to return transitions. Their main role is to separate control and return type of the *normal* execution sequence from the *exceptional* one.

Object creation. In this case the source and destination locations are $\lambda = \beta : c_{b_k}$ and $\lambda' = \beta : r_{b_k}$ where c_{b_k} and r_{b_k} are call and return nodes respectively, of the object creation box b_k . Let d be the class and x the return variable associated with box b_k . Then $(\lambda, \alpha, \lambda') \in \delta$ with global action α defined as below where $\text{odom}(\omega) = \cup_{c \in C} \text{dom}(\omega_c)$.

$$\{ (\varphi:\sigma:\omega, \varphi:\sigma':\omega') \mid i \in (O \setminus \text{odom}(\omega)) \wedge \omega' = \omega[d \mapsto \omega_d[i \mapsto \tau_i]] \wedge \sigma' = \sigma[x \mapsto i] \wedge \varphi:\sigma:\omega \in GF_\lambda \}$$

The action α does the following: (1) it finds a fresh object identifier $i \in O$, (2) it binds i to the default attribute environment $\tau_i \in \Sigma_{A_d}$ inside ω' and (3) it binds the return variable x to the new identifier i inside σ' .

4. Concurrent Class Machines

Concurrent CMs (CCMs) are an extension of CMs with a mechanism for forking threads. Formally, CCMs are the same as CMs except that: (1) the class hierarchy must contain a class `Thread` with no attributes and one method with signature `run():void`, (2) there is an additional kind of

⁷This is false if the value of oe is null

box, namely, a *thread start box*, and (3) the main class C_i must be a subclass of Thread.

A thread start box is similar to a method invocation box except that: (1) the class d_b is Thread, (2) the associated method identifier m_b is `start` (note that `start`, like `new`, is not a method), (3) the argument expression is omitted (because `start` has no parameters), (4) the return node has no associated variable (because `start` has no return value), and (5) there is no exception node e_b (because `start` does not throw exceptions).

The semantics of a CCM $\bar{A} = ((C, \leq), B, N, C_i, \Omega)$ is given in terms of a global state machine $\bar{A}^* = (GL, GF, C_i, \delta)$ consisting of the following elements:

- A set of *location maps* \overline{GL} that are partial functions from O to GL , where $GL \subseteq B^*N$. In particular, the domain of a location map is the set of threads that have been started.
- A set of *frame maps* \overline{GF} that are partial functions from O to GF , where $GF \subseteq \Sigma_X^*$.
- A *labeled transition relation* $\bar{\delta} \subseteq \overline{GL} \times \overline{GA} \times \overline{GL}$ where $\overline{GA} = \mathcal{P}((\overline{GF} \times \Omega) \times (\overline{GF} \times \Omega))$.

The transition relation $\bar{\delta}$ is defined by interleaving the transitions of the threads. Transitions that do not go through thread start boxes have the same semantics as in a sequential CM. To capture this in the semantics of CCMs, we consider the sequential CM A obtained from \bar{A} by adding to class Thread a method called `start` that does nothing and replacing all thread start boxes with method invocation boxes that invoke this new method. We define $\bar{\delta}$ in terms of the transition relation δ of A^* . Let $\text{threads}(\omega) = \bigcup_{c \leq \text{Thread}} \text{dom}(\omega(c))$. For $c \leq \text{Thread}$, let $\text{startGL}(c) = (n_c)$, where n_c is the call node of the run method of class c (this method might be inherited) and $\text{startGF}(c) = (\sigma_c)$, where σ_c maps local variables of c .run to default values. Note that (n_c) and (σ_c) are tuples interpreted as stacks containing one element.

Consider a location map $\bar{\lambda}$ and a thread $\theta \in \text{dom}(\bar{\lambda})$. Let $\beta : n = \bar{\lambda}(\theta)$. If n is not the call node of a thread start box, then for each transition $(\bar{\lambda}(\theta), \alpha, \lambda') \in \delta$, $(\bar{\lambda}, \text{extend}(\alpha, \theta), \bar{\lambda}[\theta \mapsto \lambda']) \in \bar{\delta}$, where $\text{extend}(\alpha, \theta) = \{(\bar{\varphi} : \omega, \bar{\varphi}[\theta \mapsto \varphi'] : \omega') \mid (\bar{\varphi}(\theta) : \omega, \varphi' : \omega') \in \alpha\}$. If n is the call node of a thread start box b , there are two cases, depending on whether the thread has already been started. The target thread is $\theta_1 = oe_{\bar{\varphi}(\theta), \omega}$. Let $c = \text{classOf}(\theta_1)$. If the target thread has not been started, i.e., $\theta_1 \in \text{threads}(\omega) \setminus \text{dom}(\bar{\lambda})$, then $(\bar{\lambda}, \alpha, \bar{\lambda}[\theta \mapsto \beta : r_b][\theta_1 \mapsto \text{startGL}(c)]) \in \bar{\delta}$, where

$$\alpha = \{(\bar{\varphi} : \omega, \bar{\varphi}' : \omega') \mid \bar{\varphi}' = \bar{\varphi}[\theta_1 \mapsto \text{startGF}(c)] \wedge \omega' = \omega\}.$$

If the target thread has been started, i.e., $\theta_1 \in \text{dom}(\bar{\lambda})$, then the `start` operation is a no-op that returns normally (we could easily adopt a Java-like semantics in which `start` throws an exception in this situation), and $(\bar{\lambda}, \iota, \bar{\lambda}[\theta \mapsto \beta : r_b]) \in \bar{\delta}$, where ι is the identity relation on $\overline{GF} \times \Omega$.

5. Trace Semantics and Refinement

Trace semantics. Given a CM \bar{A} denote the associated GCM by \bar{A}^* . A pair $(\bar{\lambda}, \bar{\eta})$ where $\bar{\lambda} \in \overline{GL}$ and $\bar{\eta} \in \overline{GF} \times \Omega$ is called a *configuration* of \bar{A}^* . We write $((\bar{\lambda}, \bar{\eta}), (\bar{\lambda}', \bar{\eta}')) \in \bar{\delta}$ if $(\bar{\lambda}, \bar{\alpha}, \bar{\lambda}') \in \bar{\delta}$ and $(\bar{\eta}, \bar{\eta}') \in \bar{\alpha}$. An *execution* of \bar{A}^* is a sequence $(\bar{\lambda}_0, \bar{\eta}_0) \rightarrow (\bar{\lambda}_1, \bar{\eta}_1) \rightarrow \dots \rightarrow (\bar{\lambda}_n, \bar{\eta}_n)$ such that for all i , $((\bar{\lambda}_i, \bar{\eta}_i), (\bar{\lambda}_{i+1}, \bar{\eta}_{i+1}))$ is an environment or a $\bar{\delta}$ step.

An *environment step* occurs when the environment invokes a method of an object shared by the CM and its environment; these objects (e.g. static shared attributes and input/output streams) form the CM's *communication interface* (CI). The net effect of the step is to push a stack frame with two local variables, *ret* and *exc*, non-deterministically assign an arbitrary value to one of them, and return that value. The CI may vary over time and can be computed inductively using an approach related to [9]. For space reasons, this computation is deferred to the full paper.

Given an execution ex its associated *trace* tr captures only the observable part of ex . The location and frame stacks are private, so they can be discarded, along with the private objects. Hence, tr contains at each i the projection of ω_i to the objects accessible from CI_i . The first and last elements of the trace also contain the call and return values. The *executions* and *traces* of \bar{A} are defined to be the same as for \bar{A}^* . The set of traces of \bar{A} is denoted $L_{\bar{A}}$.

Refinement. The trace semantics leads to a natural notion of refinement between CCMs: an implementation CCM I *refines* a specification CCM S , denoted $I \preceq S$, if $L_I \subseteq L_S$ modulo an isomorphism of object identifier names. The isomorphism is necessary, because the exact policy of allocating identifiers should be hidden and therefore not influence refinement. If the implementation CCM is large, we would like to decompose the refinement task into simpler subtasks. For this we provide two *compositional* and two *least fixpoint* induction rules. We write $C[M]$ if CCM C contains a box referring to CCM (method) M .

Theorem 1 *If $M \preceq N$ then $C[M] \preceq C[N]$. If $C[\cdot] \preceq^* D[\cdot]$ then $C[M] \preceq D[M]$. If $C[N] \preceq N$ then $(\mu x.C[x]) \preceq N$. If for all x , $x \preceq N$ implies $C[x] \preceq N$ then $(\mu x.C[x]) \preceq N$.*

The first compositional rule is justified by the monotonicity of the corresponding GCMs. The second compositional rule is justified in addition by observations at the “inner environment” border (\preceq^* relates trace sets where the call/return information at the inner box is made visible). The fixpoint rules are justified by the continuity of the GCMs. These rules are related to the rules given in [3].

6. Conclusions

We have presented CCMs, a comprehensive, machine-based model of multi-threaded, object-oriented systems. We have equipped CCMs with a visual design notation and

a collection of refinement proof-rules supporting compositional reasoning. Although not described in this paper due to space limitation, we have also studied important new techniques for generating efficient code from CCM specifications, by applying incrementalization [15, 18, 13, 12] to multi-threaded programs and to object-oriented programs; we have also studied the use of other analyses and optimizations, e.g., [11, 6, 16, 6], in our framework.

We have a prototype implementation that allows interactive specification in Visual CM and automatic generation of Java code for most features of our language. We have used the system for the specification and code generation of example applications, including the readers/writers problem and a simple telephone switch system. Performance of the generated code is similar to the best handwritten code.

A variety of visual notations, formal models and analysis methods for object-oriented systems have been proposed in the literature; an extensive bibliography can be found in [17]. What distinguishes our approach from this body of work (and more recent proposals such as [1, 10, 14]) is the comprehensive nature of the Concurrent Class Machine model. CCMs capture a host of key object-oriented concepts in one machine-based model, including classes, objects, inheritance, dynamic method dispatch, multi-threading and exceptions. Moreover our refinement rules allow one to reason compositionally about multi-threaded object-oriented systems in a trace-based setting. The CCM model and accompanying refinement rules have been inspired by the work of [3, 2, 5]. These approaches are not object-oriented and hence do not cover the array of object-oriented programming concepts featured in CCMs.

For future work, we plan to complete the implementation of the code generator and optimizer for more advanced features of CCM and assess performance of our system and the generated code via further experimentation. We are also in the process of developing a model checker in the style of [4, 5] that supports both enumerative and symbolic invariant and refinement checking of CCM models and that uses static analysis techniques similar to [8].

References

- [1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, NY, 1996.
- [2] R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *Proc. of CAV'01, Conference on Computer Aided Verification*. Springer Verlag, 2001.
- [3] R. Alur and R. Grosu. Modular refinement of hierarchic reactive machines. In *Proc. of POPL'00, the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 390–402. ACM Press, 2000.
- [4] R. Alur, R. Grosu, and M. McDougall. Efficient reachability analysis of hierarchical reactive machines. In E. Emerson and A. Sistla, editors, *Proc. of CAV'00, the 12th International Conference on Computer Aided Verification*, pages 280–296. Springer, LNCS 1855, 2000.
- [5] T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proceedings of 7th International SPIN Workshop on Model Checking of Software (SPIN 2000)*, Aug. 2000.
- [6] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *ACM Conference on Object-Oriented Systems, Languages and Applications*, pages 35–46. ACM, New York, Oct. 1999.
- [7] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.
- [8] J. C. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd International Conference on Software Engineering (ICSE)*, June 2000.
- [9] R. Grosu and K. Stølen. Stream based specification of mobile systems. To appear in *Formal Aspects of Computing*, Springer Verlag, 2001.
- [10] D. Harel and E. Gery. Executable object modelling with statecharts. *IEEE Computer*, 30(7):31–42, 1997.
- [11] C. Heitmeyer, J. Kirby, and B. Labaw. Tools for formal specification, verification, and validation of requirements. In *Proceedings of 12th Annual Conference on Computer Assurance*, June 1997.
- [12] Y. A. Liu. Efficiency by incrementalization: An introduction. *Higher-Order and Symbolic Computation*, 13(4):289–313, Dec. 2000.
- [13] Y. A. Liu and S. D. Stoller. Loop optimization for aggregate array computations. In *Proceedings of the IEEE 1998 International Conference on Computer Languages*, pages 262–271. IEEE CS Press, Los Alamitos, Calif., May 1998.
- [14] A. Mitschele-Thiel. *Systems Engineering with SDL*. J.W. Wiley, 2001.
- [15] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3):402–454, July 1982.
- [16] R. Sethi. A note on implementing parallel assignment instructions. *Information Processing Letter*, 2:91–95, 1973.
- [17] S. Webster. An annotated bibliography for object-oriented analysis and design. *Information and Software Technology*, 36(9):569–582, 1994.
- [18] D. M. Yellin and R. E. Strom. INC: A language for incremental computations. *ACM Trans. Program. Lang. Syst.*, 13(2):211–236, Apr. 1991.