DESIGNING MULTITHREADED SOFTWARE BASED ON CONCURRENCY IN THE PROBLEM DOMAIN

BO I. SANDÉN

Colorado Technical University
4435 N Chestnut St, Colorado Springs,
CO 80907, USA

(received: 3 June 2021; revised: 07 August 2021;
accepted: 8 August 2021; published online: 30 October 2021)

Abstract: *Event-sequence modeling* is a thread-architectural style for event-driven software. It bases the set of threads in a multithreaded program on an *event-sequence model* of the problem domain. Each event sequence is a time-ordered set of event occurrences in the domain. (It is often defined by a state machine.) An event-sequence *model* is a *set* of event sequences that together cover all relevant event occurrences in the domain. Occurrences in one event sequence are generally concurrent with those in other sequences. The event-sequence modeling approach leads to architectures consisting of threads, each based on an event sequence, and shared objects. The threads can run concurrently on different cores/processors except when they must have exclusive access to some shared object. This paper defines these concepts and illustrates them with examples.

Keywords: concurrency; event-driven software; event sequence; reactive software; thread architecture; threading

DOI: https://doi.org/10.34808/tq2021/25.2/c

1. Introduction

Nowadays, many computers have multiple processors or cores. This means that a single program can consist of many threads executing in parallel. For example, each thread can be analyzing a different part of some large data set. Given the abundance of cores and processors, some kinds of applications are using threads for the first time. Other, already threaded software is undergoing a generation shift from single- processor, single-core implementations to multiprocessors [1]. This may be particularly true for *control software*, which runs physical plants. "Big data" is a newer field that can take advantage of multiple processors/cores to analyze large datasets.

Transaction-control systems giving large numbers of users simultaneous access to a database illustrate threading well. Each user's transaction may involve multiple database records or tuples. Because each transaction basically runs the same logic, the complexity of the overall system is kept under control. In most cases, different transactions running at the same time don't need to access the same data and can execute independently using separate threads. We must however enforce mutual exclusion for when two simultaneous transactions do affect the same data.

We are here dealing with the software in a single computer with a physical clock and refer to the structure of threads and shared objects as a system's *thread architecture*. We shall discuss *event-sequence modeling* as a *thread-architectural style* for the design of event-driven software. It bases the threads on concurrency inherent in the problem domain.

"Problem domain" refers to that part of the real world with which the software interacts. The software may be embedded in a physical environment with devices such as robot arms operating concurrently. A dedicated thread can *software-enhance* a physical robot and secure for it exclusive access to shared physical resources. For example, the robot may need exclusive access to a certain conveyor in order to pick a workpiece.

In order to develop a thread-architectural style, we can look for inspiration to object-oriented analysis and design, where we identify classes of objects in the problem domain and their relationships and then create corresponding software structures. That way, we achieve a kind of isomorphism: The problem domain and the software solution both consist of objects, which expose certain operations. We want a thread architecture to be similarly isomorphic with the problem domain. It may, however, not be immediately obvious what in the problem domain corresponds to a thread.

In control software, a single thread can process a sequence of events one by one as they occur. An *event* usually triggers a transition between states. The "click" that causes a garage door to start moving is an example. It can have multiple unique *occurrences*, each at a specific time but without duration.

In a different example, a physical robot may signal to a dedicated, "enhancing" thread that it has picked or placed something. We can describe the robot's "life" as a sequence of event occurrences such as *"picked – placed – picked – placed – …,"* which we shall call an *event sequence*. We base each thread on such a sequence of occurrences in the problem domain. If two events can occur at the same time, i. e., *concurrently*, then two different threads should process them, often running on different processors or cores.

A set of event sequences that partition the set of relevant occurrences in a given problem domain form an *event-sequence model* of that domain. Such a model reflects how concurrent the problem is. On that basis, we can identify an optimal set of threads and avoid making the software more or less concurrent than what is useful.

2. Related research

2.1. Control systems software

A control system manages some process in the real world via sensors and actuators. A "hard real-time system" has conflicting, hard deadlines. A computation has a hard deadline if it must complete by a given time to avoid dire consequences. Deadlines *conflict* if threads must be scheduled in a particular order to ensure that they *all* meet their deadlines. The oldest software technology for hard real-time problems is the *cyclic executive*, which does not involve threading [2].

Replacing the cyclic executive with a set of periodic threads can make the software more easily modifiable. With appropriate thread scheduling and on certain conditions, such a system can be as predictable as a cyclic executive. Priority-based scheduling is, for example, standard practice in automotive systems programming [3].

Rate-monotonic scheduling (RMS) is a well-known policy that lets us determine ahead of time whether a set of threads performing periodic computations with known execution times will *all* meet their periodic hard deadlines. With RMS, each thread's priority is a function of its period – the shorter the period, the higher the priority [4]. The scheduling is *preemptive*: A thread that is ready to run can take over a processor from a lower-priority thread.

Interesting issues are *priority inversion* and *push-through stalling*. Priority inversion exists, for example, if a thread, *t1*, is ready to run but needs exclusive access to a shared resource held by a lower-priority thread, *t2*. Then *t1* must let *t2* proceed and release the resource. On a single processor, a third thread, *t3*, with intermediate priority, which does not need access to the resource, can create *push-through stalling* by preempting *t2* and thereby prolonging *t1's* wait. In turn, *t3* may also be preempted, leading to a generally unpredictable delay for *t1*. The usual remedy is to raise *t2's* priority while it holds the resource. These issues are of theoretical as well as practical interest, which means that some systems where they occur are carefully analyzed, designed and documented [5].

In the end, however, thread scheduling is all about husbanding limited processing power. When an abundance of cores and processors makes this less necessary, threads can complete their computations on time without special scheduling efforts. Priority inversion remains an issue while push-through stalling may become less important as threads are likely to find idle processors when necessary [6].

2.2. Thread-design methodology

Gomaa [7, 8] provided a systematic analysis and design approach for reactive software. It is primarily focused on data flow, and the software system is first represented as "a collection of collaborating objects that communicate by means of messages" (Gomaa [7], p. 306). In the analysis of a control system for an elevator bank with multiple cabins traveling in parallel shafts, [7] defines a number of "use

cases" such as *Stop Elevator at Floor*. A collaboration diagram (Gomaa [7], Figure 18.7, p. 471) illustrates this use case in terms of objects exchanging messages. It includes a "state dependent control object" of class Elevator Control, which encapsulates a state machine (Gomaa [7], Fig. 18.13, p. 479).

UML collaboration diagrams drawn for the various use cases are consolidated into a single diagram (Gomaa [7], Figure 18.14, p. 481), which shows the interactions between objects in the elevator control system as a whole. After subsystem structuring, the analyst determines which objects may execute concurrently. A single diagram shows a static model of the thread architecture for an entire system such as an *Elevator Control System* (Gomaa [7], Figure 18.18, p. 486).

Then follows a structuring of the system into threads according to criteria laid out in Chapter 14 (Gomaa [7], pp 305-360). With multiple elevator cabins in parallel shafts all controlled by a single computer, each cabin has its own instance of class Elevator Control (Gomaa [7], p. 488). Each such instance then becomes a *thread* of class Elevator Controller, again because its logic is based on a state diagram.

A collaboration diagram of the thread architecture of the Elevator Control System (Gomaa [7], Figure 18.19, p. 490) shows multiple threads of class Elevator Controller as well as the two singleton "coordinator threads," Elevator Manager and Scheduler. Elevator Manager adds internal requests from buttons inside cabins to a list of committed stops, which is kept in each cabin's Elevator Status & Plan object. The Scheduler thread processes each floor-button request as soon as it's made, "selects the most appropriate" cabin to handle it (Gomaa [7], p. 491), and updates that cabin's list.

Each Elevator Controller thread consults the list in its own Elevator Status & Plan object to find out which call to serve next. There are some additional interactions; for example, Elevator Manager informs appropriate, idle Elevator Controller threads of new requests to serve.

2.2.1. Discussion

In sum, the approach generates many diagrams, building on each other. It determines that something is a "state dependent control object" (Gomaa [8], pp. 152-153) if it executes a state diagram. It later makes it a "state-dependent control thread" for the same reason (Gomaa [8], p. 245). (It is unclear while this alone justifies a thread.)

The determination of which objects may execute concurrently happens at quite a detailed level. In the implementation of the use case *Request Elevator*, for example, an elevator-button interface thread, which picks up the button requests, sends each request to an Elevator Manager coordinator thread, which updates an Elevator Status and Plan object. Similarly, in the implementation of the "Stop Elevator at Floor" use case, an "asynchronous input device interface" thread notifies the state-dependent control thread Elevator Control that the arrival sensor was tripped (Gomaa [8], p. 487-488).

The effect is a number of unique and specialized threads that are closely intertwined. To describe how a request is handled, you must summarize one thread after the other. The design approach focuses on data flow where each request travels from a button to the list of planned stops for a particular cabin. The cabin then stops at different floors according to its list. (It's somewhat unclear in which order the listed floors are visited.)

While some threaded systems involve data flow, many others do not. Even when there is an element of data flow, it may not be a particularly important aspect. A more general starting point is to look for aspects of the problem itself that lend themselves to being viewed in terms of concurrency. This can be done before any decomposition into modules. Thus, in the multi-cabin elevator system, the cabins operate concurrently. The difficult part is to make each cabin behave predictably and efficiently while all cooperating to serve the requests. For this reason, rather than focusing on the data flow, we must address the behavior of an elevator cabin early on to ensure that it moves in a predictable way, stops when necessary, and not too often when it's not.

Rather than transporting each new request all the way into a specific cabin's Elevator Status & Plan object, a more straightforward approach would be to keep outstanding requests in a shared data structure, which each Elevator Controller thread can query when its cabin is approaching a floor, update after visiting a floor and consult when deciding whether to continue up or down or to turn around. (See also 6.2)

3. Thread architectures

The elements of a thread architecture are of two kinds [9]: On the one hand, there are the *threads*, defined as "independent paths of execution through program code" [10]. Threads are fundamentally intended to run concurrently on separate cores or processors, but a single core or processor can also handle multiple threads. A language's run-time system or an operating system provides the threads. Information normally passes between threads asynchronously, via safe objects.

Safe *objects* are called *synchronized* objects and classes in Java [11] and Monitor Objects in other languages [12]. A thread can *lock* a safe object, and thus obtain exclusive access to it for a short time, typically milliseconds. This is referred to as *exclusion synchronization*.

A safe object can also provide *condition synchronization* and block calling threads until some condition holds. This allows us to create *semaphore safe objects* that safeguard a resource in the problem domain by granting one calling thread at a time exclusive access to the resource. The calling thread may need exclusive access to such a domain resource for minutes. A semaphore safe object exposes an operation *Acquire*, which blocks the calling thread until its turn comes to get

exclusive access, and the procedure *Release*, which the thread calls when it no longer needs the exclusive access.[1]

A semaphore safe object can also represent a set of *interchangeable* resources and give each calling thread exclusive access to any one of them. *Acquire* then returns the identity of the allocated resource in an output parameter while *Release* expects the identity as an input parameter.

Semaphore safe objects form a subclass of the broader class of *state-machine safe objects*. Whereas a semaphore safe object usually has two states such as *Free* and *Occupied*, and two operations, such as *Acquire* and *Release*, a state-machine safe object may have multiple states and operations.

3.1. Example

Saez [5] presented a small but quite realistic *pick-and-place* problem that nicely illustrates concurrency, much of which is plain to see in the problem domain. The problem is not trivial but quite easy to understand. As Fig. 1 shows, Robot0 and Robot1 pick workpieces of two different types off the *input conveyor* belt to the right and place each piece by type on one of two *output conveyors* to the left marked "Classified output."

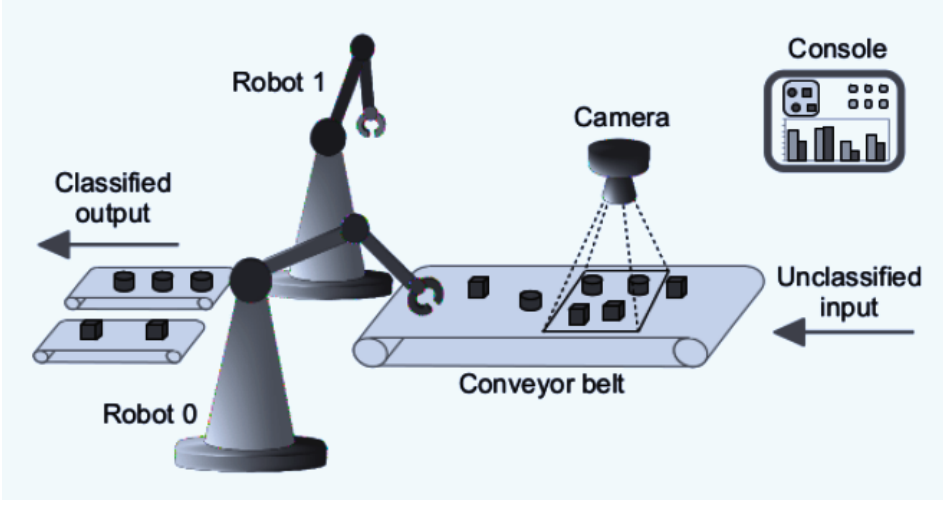

Figure 1. Pick-and-place system [5]

As pieces arrive on the input conveyor, a video camera captures successive frames. An image-processing routine called *segmentation* determines the number of pieces and their positions in each frame and inserts this information into an *ImageBuffer*. Based on that, a *recognition* routine determines each piece's type and its position and orientation on the conveyor. It makes this information about each piece available to the robots in a *workpiece record*. Based on such a workpiece record,

1. A recent discussion of such fundamental concepts in concurrency can be found in [13].

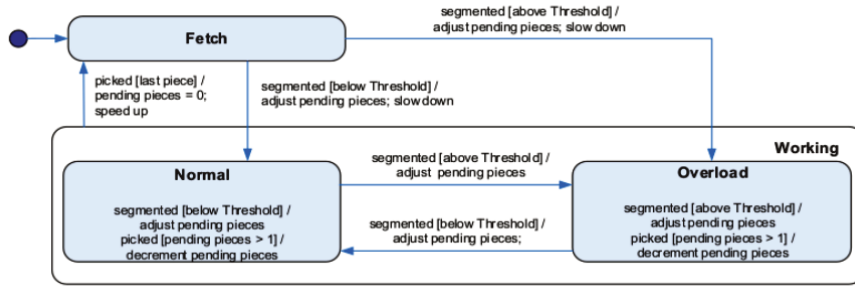

Figure 2. Global state diagram of the pick-and-place system. "Pending pieces" is the number of segmented pieces yet to be picked

a robot can locate the piece on the conveyor, pick it, and place it according to its type on one of the "classified output" conveyors. Each robot is autonomous but sometimes needs exclusive access to a shared resource such as a conveyor.

At its global level, the pick-and-place system has the states shown in Fig. 2. It starts in *Fetch*, where the input conveyor moves at its higher speed. As soon as *segmentation* detects a nonempty frame, the system transitions to superstate *Working*, where the conveyor moves more slowly. The system enters substate *Normal* if the frame's piece count is below a certain threshold, and *Overload* otherwise. The system toggles between those two substates depending on the piece count in each frame. When there are no pending pieces, the state changes to *Fetch* and the conveyor speeds up.

In state *Normal*, only Robot0 picks, and *segmentation* and *recognition* process a frame completely by the time the next one arrives. In *Overload*, both robots pick, and the segmentation and recognition of consecutive frames overlap. The robots must pick workpieces approximately in the order they appear on the conveyor so that all workpieces in one frame can be picked before any of those in the next frame.

3.1.1. Software enhancement of physical devices

Reactive software such as in the pick-and-place problem must often give domain-resource users, such as robots, exclusive access to shared domain resources, such as conveyors. A robot may need the conveyor for several seconds or minutes. We let a thread, *Robot*, say, software-enhance the resource *user* and call the operations Acquire and Release on a semaphore safe object associated with the conveyor.

The *Robot thread* and the physical robot may cooperate so closely that we can often think of both together as one software-enhanced device: The physical robot does the actual picking and placing while its software-enhancing *Robot thread* secures exclusive access to shared resources and may also, for example, compute when and where a part is to be picked. The robot might signal to its

Robot thread, "I'm at the picking position." The thread then works out the timing and gives the physical robot the command "pick now."

4. Event-sequence modeling[2]

Event-sequence modeling[2] is an approach to the design of thread architectures based on concurrency found in the problem domain. While a thread architecture consists of threads and safe objects, identifying the threads is often most important.

We shall use the term ***event sequence*** to describe a time-ordered set of event occurrences in the problem domain, which the software must process one by one. No two occurrences in an event sequence are at the same time. Generally, we base each thread in the software on such an event sequence.

As we base each thread on an event sequence, we base a whole thread architecture on an *event-sequence model* of the problem domain. Such a *model* captures a problem's concurrency as a set of event sequences. The model leads to a multi-thread architecture that is as concurrent as the problem itself. We thus use concurrency identified in the problem domain to determine what should be concurrent in the software.

The following subsections focus on the analysis of problem domains and expand on the concepts of event sequences and event-sequence models.

4.1. Event sequences defined

Formally, an event sequence consists of event occurrences totally ordered under a relation called *"before"* [18]. We shall say that any two event occurrences, x and y, have a *before* relationship if we know that, in our problem domain, *x is before y.* The situation where one thread sends a message to another is a defining example: The event that the message is sent always occurs before the event that it is received.

For any two occurrences, x and y, in an event sequence, either x is $before$ y, which we can also show as: $x \rightarrow y$ or y is $before$ x: $y \rightarrow x$. If neither $x \rightarrow y$ nor $y \rightarrow x$ holds for the two occurrences, then they are said to be concurrent. We don't know the order of such concurrent occurrences a priori; they may be at the same time. No event sequence can include concurrent occurrences, but occurrences in different event sequences are usually concurrent.

In the pick-and-place problem, for example, each robot operates sequentially and generates a *Robot* event sequence. A fragment of it may be: *input conveyor locked – arrived at picking position – picked – arrived at input holding area – input conveyor unlocked – arrived at output holding area – output conveyor locked – arrived at placing position …* The *Robot* event-sequence *type* has two instances, *Robot*0 and *Robot*1, which differ slightly.

2. Event-sequence modeling builds on earlier work on entity-life modeling, see Sandén ([14], [15], [16]) and [17].

In the pick-and-place problem, we can express the processing of successive frames, too, in terms of event sequences:

- The event *picture taken* is that the image of a frame becomes available. It triggers *segmentation*, which detects the number of workpieces and their positions in the frame.
- The event *segmented* is that a freshly segmented frame becomes available in an image buffer. This triggers the *recognition* of each piece in the frame, which determines the piece's type and its location and orientation on the conveyor.
- The event *recognized* is that a workpiece *record* with the above information becomes available to the robots.

Thus an event-sequence called *Frames* could be: *"picture taken – segmented – recognized – recognized – … – picture taken – segmented – recognized, …"* Unlike, say, *picked*, which is an external event that the software must wait for, the software itself makes the events *segmented* and *recognized* occur. As long as they are ordered relative to the other events, we can include them in event sequences. Thus, we fit the occurrences of all relevant events in the problem into event sequences even if the software creates some of the occurrences. It is a consistent way of capturing the concurrency in a problem and ultimately implement it in software.

4.2. Event sequences and state machines

Although we often define an event sequence in terms of a state machine, event sequences have a practical advantage: Because they are ordered sets, we can manipulate them much more easily than state machines. For example, we can *merge* – i.e., form the union of – two event sequences if that union includes no concurrent occurrences. We can also break an event sequence into *fragments*, each of which is an event sequence. The details should be defined by a state diagram or in some other formal notation. Some practically useful event sequences *cannot* be defined by state machines. They may be recursive as in the case of tree traversal, where each node in a tree-shaped data structure is visited once.

A state machine such as Fig. 2 – minus any activities – defines an event sequence. We call this particular one the *Global* event sequence for the pick-and-place problem. Its only events are *picked* and *segmented*, and a fragment of it is: *"segmented – picked – picked – picked – picked – … – segmented – picked – picked – …"*

The internal structure of an event sequence can include variants. For example, a soda machine may support the following event-sequence fragments:

- *money inserted – soda button pressed – soda dispensed*
- *money inserted – soda button pressed – soda dispensed – change returned*
- *money inserted – change-return button pressed – change returned*

Each successive customer executes *one* of these fragments. The soda machine goes through this process repeatedly for a series of customers, one after the other. All this fits into a *single event sequence*. The *implementation* can have a single thread

with statements such as "*if* sum inserted is greater than the price, *then* return change." *Additional* event sequences are only needed if the machine can serve more than one buyer *at a time*. If so, the problem is concurrent: Each event sequence would serve its own series of buyers and would be implemented as a separate thread. Together, those event sequences would make up an event-sequence *model* of the vending-machine problem.

4.3. Event-sequence models

In many problem domains, we can find a variety of event sequences, some of which may *intersect* in the sense that a given occurrence is in more than one sequence. We cannot design a thread architecture based on just any set of such event sequences. Instead, we base it on an **event-sequence *model*** of a particular problem domain. Here is a definition:

An event-sequence model of a problem domain is a set of one or more event sequences that partition the set of all relevant occurrences in that domain.

We refer to those event sequences as the *members* of that event-sequence model. They do *not* intersect; each event occurrence in a certain problem domain is part of *exactly one* of the event sequences in a model of that domain. We usually implement each such member of the event-sequence model as a thread, which processes each occurrence in its event sequence.

An event-sequence model is intended to be *restrictive* to ensure that we can implement it in a thread architecture. We base a thread on each member of an event-sequence *model* so that each event occurrence will have one designated thread handling it.

4.3.1. An event-sequence model of the pick-and-place problem

The pick-and-place problem in Fig. 1 has an event-sequence model with the members *Robot*0, *Robot*1, $Frames0$ and $Frames1$. The state machine *Global* (Fig. 2) also defines an event sequence, but notably, we cannot fit it into this model as every occurrences of *picked* is in a *Robot* event sequence, and each occurrence of *segmented* is in a *Frames* event sequence: The *Global* event sequence thus *intersects* with the sequences *Robot* and $Frames$. Keeping the *Robot* and the $Frames$ event sequences as members of the event-sequence model and excluding *Global* is our only choice: The *Robot* and $Frames$ sequences can co-exist in one event-sequence model because they do not intersect, while the *Global* event sequence can co- exist with neither.

4.3.2. Trimming an event-sequence model

We must also guard against too many event sequences, which may lead to redundant threads. Ideally, an event-sequence model should have no more members than the greatest number of events that can ever occur at the same time. It is not always possible to determine that number, however.

To trim an event-sequence model, we can *merge* some of its members. Members that could – and should – be merged into a single one are said to be *mergeable*:

Two members of an event-sequence model are mergeable if and only if their union contains no concurrent occurrences.

In the pick-and-place problem, for example, we might first identify i) an event sequence consisting of all the occurrences of *picked*0, which is where *Robot*0 picked a piece, and ii) another event sequence of all the occurrences of *placed*0, which is where it placed a piece. These two event sequences are mergeable because the robot cannot pick and place at the same time. On the other hand, the two event sequences *Robot*0 and *Robot*1 are *not* mergeable as one robot can pick while the other places.

4.4. Event sequences and threads

In principle, every thread in the software architecture should process one *event sequence* by handling one event occurrence after the other. That is the reason why we identify event sequences. A single, long-lived event sequence may actually require a succession of threads, however, perhaps one running today and one tomorrow. It may also be that the software crashes and then restarts with a new set of threads, while the event sequences just keep going. In multitier client-server architectures, the presentation, application processing and data management functions may be physically separated into tiers so that an event occurrence is processed by a different thread in a different computer for each tier. While we have defined "event sequences" formally, the concept is very practical. For example, it's quite obvious that a robot moves in discrete steps and thus creates an event sequence. Similarly, someone entering data on a computer inevitably creates an event sequence of separate key-ins, one after the other.

5. Design and implementation

5.1. Design of the thread architecture

An event-sequence model forms the basis of a *thread architecture*, which consists of threads and safe objects. It is a critical aspect of any multithreaded software. The thread architecture is self-contained and complete because every instruction must be executed by a thread. At the same time, it is more compact than the full software architecture.

We base each thread in the architecture on one member of the event-sequence model of the problem domain. Thus, for the pick-and-place problem, we build a *Robot* thread on each *Robot* event sequence and a *Frames* thread on each *Frames* event sequence.

A *Frames* thread performs the *segmentation* and *recognition* of successive frames and produces *workpiece* records. Each *workpiece* record must then be transferred to a *Robot* thread. This is done via a queue of records called *ToDo* [5].

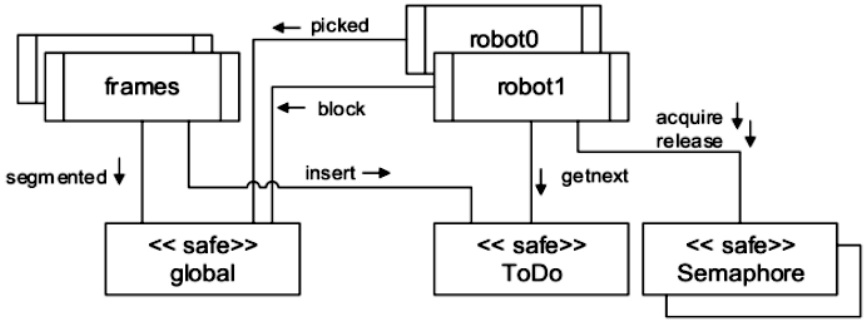

Figure 3. Thread-architecture diagram of the pick-and-place system

Each *Frames* thread inserts a record in *ToDo* for each newly recognized workpiece while each *Robot* thread repeatedly retrieves a record.
Fig. 3 shows the thread architecture of the pick-and-place problem as a call hierarchy. Boxes with double sides represent threads and are on top. Safe objects are marked with the stereotype ≪safe≫. *Robot* threads, which software-enhance the physical robots, acquire and release exclusive access to domain resources such as conveyors. For this reason, we associate a semaphore safe object with each conveyor. *Robot* and *Frames* threads also call operations on the *Global*-state-machine safe object.

A thread-architecture diagram such as in Fig. 3 shows how the pick-and-place problem works in terms of the interactions between threads and shared objects. It can be a roadmap for anyone having to study the code in some detail.

5.2. Designing a thread for each member of an event-sequence model

The event-sequence model consists of one or more event sequences such that every event occurrence in the problem domain belongs to exactly one. Each event sequence is a series of discrete occurrences over time. Generally, we give each of those event sequences a thread in the software architecture, which deals with each occurrence immediately and may run on its own core or processor.

If the event sequence is based on a state model, we typically base the thread's internal logic on it. A thread that software-enhances some physical device (such as a robot), must keep track of the state of the entire enhanced device including the physical part as well as the enhancing software.
Fig. 4 is a state diagram of the software-enhanced Robot0. It shows a loop where the robot gets a record from *ToDo*, picks the corresponding workpiece from the input conveyor, places it on an output conveyor, and again gets a record from *ToDo* possibly after a wait. Some states, such as those where the robot is moving, refer to the physical robot. But other states are directly meaningful only to the enhancing software. This includes the superstates where the robot has exclusive access to a conveyor.

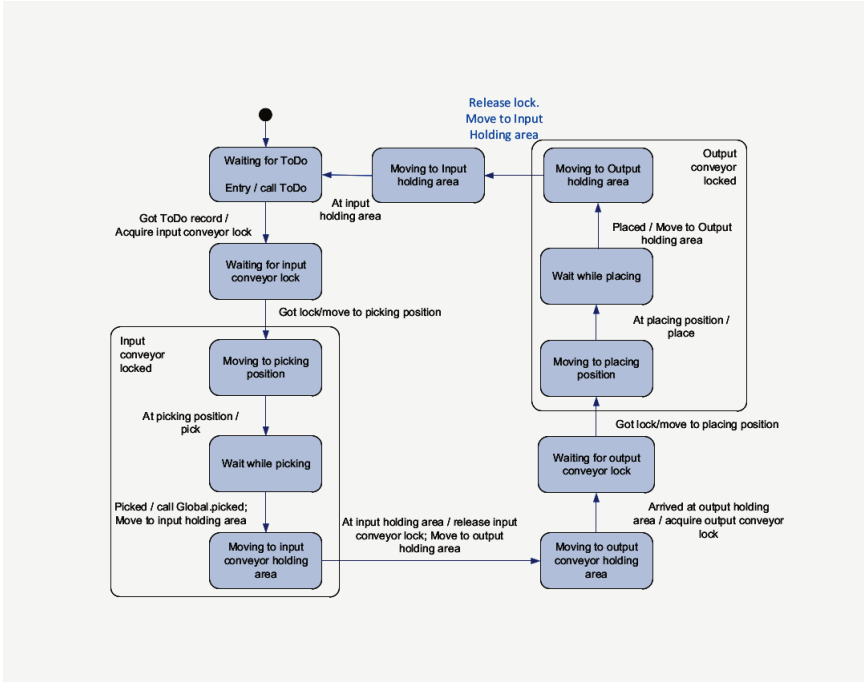

Figure 4. State diagram of Robot0 in the pick-and-place problem

Event-sequence modeling does *not* prescribe how to implement individual threads. It is, however, important to illustrate how the diagram translates readily into code for a *Robot* thread. Here is a possible implementation, which represents the state implicitly, meaning that there is no state variable indicating the current state. (The only difference between the threads *Robot*0 and *Robot*1 is that *Robot*1 blocks unless the state is *Overload*.)

loop
–*Assertion: Robot is at its Input-Conveyor holding area*
If this is *Robot*1, block until the global state = *Overload*
Get next work-piece record from *ToDo* – The thread may block here
Acquire Input-Conveyor *lock*
Move robot to picking position – Critical section start
Pick workpiece
Call *Global.Picked*
Move robot to Input-Conveyor holding area
Release Input-Conveyor lock – Critical section end
Move robot to Output-Conveyor holding area
Acquire Output-Conveyor lock – Critical section start
Move robot to placing position
Place workpiece
Move robot to Output-Conveyor holding area
Release Output-Conveyor lock – Critical sections end

Move robot to Input-Conveyor holding area
end loop

Such implicit state representation produces code that reads from top to bottom as in this example. No state variable is usually needed. Each *Acquire* - *Release* pair brackets a *critical section*, which is a code sequences accessible to one robot thread at a time. The robot is in a particular state precisely when it is executing such a critical section, so that state is inherently implicit.

5.3. Designing and implementing state-machine safe objects

In addition to the state maintained by individual threads, there are state machines that multiple threads must access such as the *Global* state machine in the pick-and-place problem. We implement a state machine that is not local to a single thread as a *state-machine safe object*. Such a safe object must use a *state variable* to keep track of the current state. It cannot represent the state implicitly as it needs to preserve it between calls. We refer to this *as explicit* state representation. A state-machine safe object can have safe *operations* such as the following:

- *Event handlers,* which change the state and/or take actions when prompted by events occurring
- *State queries,* which return the current (super)state
- *State-wait operations,* where threads block – if necessary — until a certain (super)state is entered. For example, *Robot*1 blocks until state *Overload* is entered.

A thread calls an *event handler* while it is processing an event occurrence affecting a state-machine safe object in some way. A thread calling a *state query* must usually be at a place in its program logic where it can act on a state change directly. Similarly, it should call a *state-wait operation* only when in a situation where it can safely remain until the state changes. For example, the *Robot*1 thread must ensure that its physical robot is out of the other robot's way before calling a state-wait operation.

5.4. Representing global state

Typically, threads have to adjust to global-state changes, but because each thread is quite autonomous, we can often avoid forcing them all to act on a global state change at once. Forcing a state change on a thread that is not at a convenient place in its logic can require special syntax and complicate the code.

In the pick-and-place system, for example, the physical Robot1 picks and places workpieces only in state *Overload*. It can also *place* a piece in *Normal* or *Fetch* but does not *pick* in those states. Once Robot1 has placed a piece and is at rest at a holding position, its enhancing thread *Robot*1 calls a state-wait operation on *Global* and blocks if necessary until the state is *Overload*. (The thread *Robot*0, on the other hand, works the same in all states.)

In the thread architecture (Fig. 3), the *Frames* and *Robot* event sequences become threads, which means that each occurrence of *segmented* and *picked* has

a thread to process it. Each event occurrence is thus handled from beginning to end by a single *thread* even if the handling logic is partly defined in a state-machine safe object.

In the state machine in Fig. 4 and also in the pseudocode of a *Robot* thread in 5.2, the *Robot* thread waits for some signal from the physical robot that a piece has been picked. Once the signal comes, the *Robot* thread calls the operation *Global.Picked* and then returns to the *Robot* thread's own body. In a similar fashion, the *Frames* threads call *Global.Segmented* when appropriate. While executing those calls, the threads take any action specified by the Global state machine (Fig. 2) such as speeding up the input conveyor when there are no pieces to pick.

The *Global* event sequence is *not* a member of an event-sequence model and does not have its own thread in the implementation. Such a thread would only get in the way. The *Robot* and *Frames* threads would have to communicate with it whenever an event occurs, perhaps by messages. This would incur unnecessary complication and overhead.

6. Elevator-bank example

This example is quite similar to the one in section 2.2. An elevator system does not manage personal use cases specifying each traveler's complete trip from one floor to another. It only keeps track of which buttons have been pressed. See also Jackson [19] and Sandén [15].

In this elevator bank, all cabins serve all floors and, between them, meet all passengers' needs. In a typical travel pattern, each cabin starts at the ground floor, travels up as far as necessary and then returns down to the ground floor, etc. Thus passengers who want to travel in the cabin's current direction expect to go straight to their destination even though the cabin may stop one or more times on its way there.

In 2.2, the multi-cabin elevator system is primarily viewed as a dataflow problem where each request from a button is allocated to a cabin as soon as it's made. But such requests cannot generally be served in the order they are made. To maintain the travel pattern that passengers expect, each cabin must always operate based on where it is and which direction it's traveling. For example, if a passenger enters an up-bound cabin at floor 5 and wants to get out at floor 6, then floor 6 becomes the next stop even if everyone else on board is going higher.

6.1. Subproblems

The elevator is an example of a problem that can be broken down into subproblems, each with its own event-sequence model. We can do this by partitioning the entire *set of events* into one *subset* per subproblem. Within each subset, we then identify event sequences and, ultimately, threads. In the elevator bank we can identify three such event subsets:

a) The events when someone presses a floor button to call the elevator. We shall call this the *External-Buttons* subproblem.

b) The events when someone presses a button inside a cabin to make it stop at a certain floor. We shall call this the *Internal-Buttons* subproblem.

c) The sequence of events happening to a cabin as it travels up and down, arriving at various floors and stopping as necessary. We call this the *Cabins* subproblem.

The *Cabins* subproblem shares the safe object *External-Requests* with the *External-Buttons* subproblem and the safe object *Internal-Requests* with the *Internal-Buttons* subproblem (Fig. 5). All three subproblems share the object *Servable-Requests*. The two *Buttons* subproblems are quite similar.

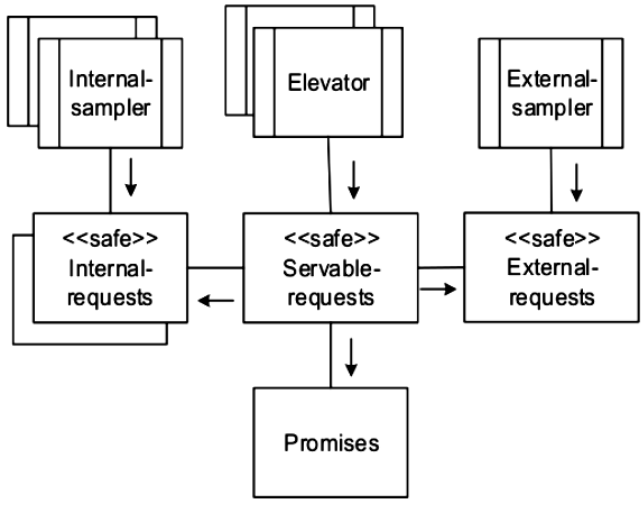

Figure 5. Thread-architecture diagram of the elevator-bank control system

All event occurrences in sets *a)* and *b)* are concurrent with those in set *c)* because a passenger can always press a button no matter where the cabin is and what it is doing. The *threads* handling the events in sets *a*) and *b*) update a data structure of outstanding calls while those in set *c*) retrieve data about outstanding calls and also mark calls as served.

6.2. The Cabins subproblem

The *Cabins* subproblem is challenging, and we might want to consider different event-sequence models of it alone. The *event-sequence model* of the *Cabins* subproblem has one event sequence called *Elevator* for each cabin. In the *thread architecture* (Fig. 5), this translates into a thread type, also called *Elevator*, which software-enhances a cabin and consults the repositories under exclusive access. Similar to the robots in Fig. 3, each cabin gets its own instance that describes the cabin's movements.

The *Elevator* threads call the safe object *Servable Requests*, which represents all requests outstanding at each point in time. The most precise way to

serve all requests in a timely fashion is to give each *Elevator* thread access to all relevant requests just before the cabin reaches a floor and just before it leaves a floor after stopping. If we instead have it act on decisions made when a request is made, the precision may suffer and the cabin may continue higher up than necessary or stop at some floors unnecessarily. *Servable_Requests* exposes a number of operations that the *Elevator* threads call:

Visit: Whenever a cabin is *closing in* on a floor, its *Elevator* thread calls the Boolean function $Visit$ to determine whether it has to stop. If $Visit$ returns the value $false$, the cabin does not stop and the *Elevator* thread calls *Passed* to remove any *promise* made to stop at this floor in this direction. (An *Elevator* thread makes a promise when it sets out to serve a particular request. The repository *Promises* in Fig. 5 keeps track of requests that cabins have committed to serve but not yet served.)

Visited: After a cabin has stopped, its *Elevator* thread calls $Visited$ to report that it has served a call. $Visited$ removes from the repository any internal call from within this cabin for this floor and any call from the button at this floor for travel in the cabin's current direction. It also removes any *promise* that this cabin has kept by stopping.

Continue_up and ***Continue_down***: When the cabin is ready to start moving, it calls *Continue_up* or *Continue_down* to find out if it must continue the way it's going. It needs to continue *up* if:

- An internal button has been pressed for a higher floor, or
- A floor button at a higher floor has been pressed. In that case, the *Promises* data structure comes into play: The *Elevator* thread ignores any call from a higher floor that another cabin has already promised to serve.

Sandén (1994, Figure 8-19, pp 351-354) shows an implementation of the *Elevator* thread with implicit state representation. The *Elevator* thread has quite a complex control structure with nested loops and if-statements to describe a cabin's behavior. We can follow the elevator cabin's movements in the code without referring to state variables (other than the floor number).

While it is possible to represent a cabin's behavior in a state diagram, each stop involves a sequence of states such as waiting to arrive at floor, waiting for doors to open, waiting for doors to close, allowing time for cabin requests, etc., which may be easier to map out in pseudocode than in a state diagram.

However we choose to implement the thread, the physical cabin performs trips from the ground floor to the highest floor necessary, and back down. There are two inner loops over floors, one upward, and one downward. If the cabin does not need to reach the top floor, it breaks out of the upward loop and starts traveling down.

This solution can accommodate any reasonable number of elevator cabins each moving it its own shaft and driven by its own *Elevator* thread. One similarity between the elevator and the pick-and-place problem is that one set of threads

collect requests for service, which the other set of threads fulfills. The two sets are connected via a safe object, which is not a simple, first-in-first-out queue. In the pick-and- place problem, the safe object *ToDo* preorders the requests to ensure that workpieces are picked roughly in the order they appear on the conveyor. This is impossible in the elevator problem. Instead, each cabin's thread determines whether to serve a request or not based on its own position and direction of travel. The requests are in a repository, not a queue or list.

7. Conclusion

Some may object to the idea that there is such a thing as a thread architecture. This is because threading has often been seen mostly as a pragmatic engineering device rather than a means for architectural structuring. However, some systems, such as the pick-and-place problem are defined by concurrency. Other software, such as transaction systems, are also fundamentally concurrent and best thought of in terms of threads and their interactions.

Just as we can base software objects on objects in a problem domain, we can decide whether a new software system should be threaded by establishing whether concurrency exists in the problem domain. We have here done this by identifying event sequences that reflect, for example, the behavior of an elevator cabin or a robot.

One desirable result of this approach is a thread architecture that is understandable. Threaded software can be harder to maintain than sequential programs. It helps if the thread architecture is clear and as simple as possible. There should be relatively few thread types, although each can have many instances. Each thread type should have an intuitively clear identity and purpose.

References

[1] Sandén B I 2018 *Designing multitask control software in a multiprocessor world*, Ada User Journal, **39** (3) 203

[2] Burns A, Fleming T and Baruah S 2015 *Cyclic executives, multi-core platforms and mixed criticality applications*, 27th Euromicro Conference on Real-Time systems, 7-10th July,

[3] Ernst R 2018 *Automated driving: The cyber-physical perspective*, IEEE Computer, **51** (9) 76

[4] L Sha, M Klein and J B Goodenough 1991 *Rate monotonic analysis for real-time systems*, Software Engineering Institute, Pittsburgh, PA, Tech. Rep. CMU/SEI-91-TR-6

[5] Saez S, Real J and Crespo A 2012 *An Integrated Framework for Multiprocessor, Multimoded Real-Time Applications*, *Springer*, In: Brorsson, M., Pinho, L. M. (eds.) Ada-Europe 2012. LNCS, Heidelberg, **7308** 18

[6] W Stallings 2015 *Operating systems: Internals and design principles*, 8 th Ed., Pearson

[7] Gomaa H 2000 *Designing Concurrent, Distributed, and Real-Time Applications with UML*, Addison-Wesley

[8] Gomaa H 2016 *Real-Time Software Design for Embedded Systems*, Cambridge University Press

[9] Sandén B I 2011 *Design of multithreaded software: The Entity-life modeling approach*, Hoboken, NJ: Wiley

[10] Friesen J 2015 *Java Threads and the Concurrency Utilities*, Apress

[11] Sandén B I 2004 *Coping with Java threads*, IEEE Computer, **37** (4) 20 doi: doi.ieeecomputersociety.org/10.1109/MC.2004.1297297
[12] Schmidt D C, Stal M, Rohnert H, Buschmann F 2000 *Pattern-oriented software architecture: Patterns for concurrent and networked objects*, Hoboken, NJ: Wiley, **4**
[13] Rajsbaum S and Raynal M 2020 *60 years of mastering concurrent computing through sequential thinking*, SIGACT News, **51** (2) 59 doi: https://doi.org/10.1145/3406678.3406690
[14] Sandén B I 1989 *An entity-life modeling approach to the design of concurrent software*, CACM, **32** (3) 330
[15] Sandén B I 1994 *Software Systems Construction with Examples in Ada*, Prentice-Hall
[16] Sandén B I 2003 *Entity-life Modeling: Modeling a thread architecture on the problem environment*, IEEE Software, **20** (4) 70 doi: doi.ieeecomputersociety.org/10.1109/MS.2003.1207459
[17] Sandén B I and Zalewski J 2006 *Designing state-based systems with entity-life modeling*, Journal of Systems and Software, **79** (1) 69
[18] Lamport B 1978 *Time, clocks and the ordering of events in a distributed system*, CACM, **21** (7) 558
[19] Jackson M A 1983 *System Development*, Prentice-Hall International

Bo I. Sandén received an M.S. in Engineering Physics from Lund Institute of Technology in 1970 and a Ph.D. in Computer Science from KTH in Stockholm in 1978. He held positions with UNIVAC and Philips Electronics in Sweden, 1971-1986. He was a visiting Associate Professor at the Wang Institute in Tyngsboro, MA, 1986-1987 and an Associate Professor of Software Engineering at George Mason University, Fairfax, VA, 1987-1996. In 1996 he joined Colorado Technical University (CTU), Colorado Springs. As a Professor Emeritus at CTU, Dr. Sandén teaches the doctoral course *Concurrent and distributed systems*. Dr. Sandén's primary research interest is the design of multithreaded software. He is the author of three books and numerous articles. His most recent book is *Design of multithreaded software: The Entity-Life Modeling approach*. Dr. Sanden is a senior member of ACM and a member of the IEEE Computer Society.