Distributed Safety Controllers for Web Services

Anders Sandholm and Michael I. Schwartzbach

BRICS[*], Department of Computer Science
University of Aarhus, Ny Munkegade
DK-8000 Aarhus C, Denmark
{sandholm,mis}@brics.dk

Abstract. We show how to use high-level synchronization constraints, written in a version of monadic second-order logic on finite strings, to synthesize safety controllers for interactive web services. We improve on the naïve runtime model to avoid state-space explosions and to increase the flow capacities of services.

1 Introduction

An Interactive Web Service consists of a global state (typically a database) and a number of distinct sessions that each contain some local state and a sequential, imperative action. A web client may invoke an individual thread of one of the given session kinds. The execution of this thread may interact with the client and inspect or modify the global state.

To alleviate laborious low-level encodings of such services, the Mawl language [6, 2] has been suggested as a high-level notation that is compiled into low-level CGI-scripts. It directly provides programming constructs corresponding to global state, dynamic document, sessions, local state, imperative actions, and client interactions. This system shows great promise to facilitate the efficient production of reliable web services.

While Mawl thus offers automatic synthesis of many advanced concepts, it still relies on standard low-level semaphore programming for concurrency control. We have designed a variation of Mawl, called Wig, on which we are currently performing a number of experiments. One of these is to synthesize the concurrency control from a high-level notation that is designed to be simple and intuitive. Our notation is based on second-order monadic logic on finite strings, M2L-Str.

As an example of a Wig service, consider the example in Fig. 1, which provides a counter for a page. The intended behavior should be clear. By default Wig provides exclusive write-access to components of the global state, but this is clearly not enough even for this simple example, where the updates of the counter variable must be atomic, which requires some sort of critical region.

Larger web services often require quite complicated concurrency control, which is hard to implement and maintain (and not the kind of issue on which most web programmers want to spend their time).

[*] Basic Research in Computer Science,
Centre of the Danish National Research Foundation.

```
service {
    global counter: int = 0;
    document ThePage { You are visitor number <var name="num">. };
    session ReadMe {
        counter:=counter+1;
        show ThePage[num ← counter]
    }
}
```

Fig. 1. A simple Wig service.

The web programming environment, with rapidly changing code, fast machines, and slow networks seems an ideal niche for a radical approach of synthesizing finite-state controllers from high-level specifications without suffering an unacceptable performance loss compared to hand-written code. This paper provides the foundations for these ideas.

2 Labeled Services

Before presenting the actual high-level notation for concurrency control we need to make one important extension to the basic language for writing service code. The high-level notation for concurrency control needs a way of referring to points in the service code. For this purpose we add the possibility of having labels in the code. As an example it might very well later turn out to be advantageous to be able to refer to, say, the beginning and the end of a critical region.

With labels in the service code, a run of a service gives rise to the sequence of labels that are passed in turn during the run. We have the basic assumption that no two labels are passed at exactly the same time. In the absence of this assumption, we should replace the word sequence by pomset, partially ordered multi-set. Later, though, we do consider independence models in order to avoid the state explosion problem and to increase parallelism.

In addition to the labels added by the programmer, the following labels are generated automatically because standard safety requirements almost always involve these labels.

- For each global variable X we generate the labels take-X and give-X. These labels are put in just before and just after each assignment to the global variable X. They will make us able to ensure that global variables can only be updated by one session thread at a time.
- For each session definition A we generate labels start-A and end-A. They are put at the beginning and at the end of session A, respectively.

The kind of sequences that a run of a service S gives rise to will thus be strings over the alphabet Σ_S given by

$$\begin{aligned}\Sigma_S = \mathit{labels}(S) \;\cup\; \{\,\textsf{take-X} \mid X \in \mathit{globals}(S)\,\} \;\cup\; \{\,\textsf{give-X} \mid X \in \mathit{globals}(S)\,\} \\ \cup \{\,\textsf{start-A} \mid A \in \mathit{sessions}(S)\,\} \;\cup\; \{\,\textsf{end-A} \mid A \in \mathit{sessions}(S)\,\},\end{aligned}$$

where *labels*, *globals*, and *sessions* are the functions that given a service S evaluates to the names of the labels, global variables, and sessions of S, respectively. Where obvious from the context we will drop the subscript S.

An *automaton* is a structure $A = (Q, \hat{q}, \Sigma, \rightarrow, F)$, where Q is a set of states with initial state $\hat{q} \in Q$, Σ is a set of labels, $\rightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation, and $F \subseteq Q$ the set of acceptance states. We shall use $q_1 \xrightarrow{\sigma} q_2$ as notation for $(q_1, \sigma, q_2) \in \rightarrow$.

A string $w = \sigma_0 \sigma_1 \ldots \sigma_{n-1} \in \Sigma^*$ is said to be *accepted* by the automaton A if there exists a run of A that reads the string w and ends up in an accepting state q, i.e., if there exist $q_1, \ldots, q_{n-1} \in Q$ and $q \in F$, such that

$$\hat{q} \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} \ldots \xrightarrow{\sigma_{n-2}} q_{n-1} \xrightarrow{\sigma_{n-1}} q.$$

We shall denote by $L(A)$ the *language* recognized by an automaton, i.e.,

$$L(A) = \{\, w \in \Sigma^* \mid A \text{ accepts } w \,\}.$$

One can observe that a service S induces—in a natural way—an infinite state automaton with alphabet Σ_S, transitions corresponding to passing a label during execution, and $F = Q$, where the language accepted by the induced automaton will then be the set of (finite prefixes of) possible runs of that service. We shall denote by $A_S = (Q_S, \hat{q}_S, \Sigma_S, \rightarrow_S, F_S)$ the automaton induced by S. Again we may omit the subscript S.

3 Safety Requirements

While programming a service, one often needs to make sure certain properties hold. We offer a way of synthesizing runtime controllers from static *safety requirements*. These requirements—written in a dialect of M2L-Str—can together with the service code be compiled into executable code containing a runtime system that automatically ensures that the safety requirements are met, namely by compiling the safety requirements into a runtime safety controller. For a diagram of the overall compilation process see Fig. 2.

M2L-Str is a very expressive logic in which several other logics can be encoded, e.g., interval logic and all sorts of linear time temporal logics. For an introduction to and a discussion of M2L-Str see [4]. The specific high-level notation built on top of M2L-Str for writing Wig safety requirements is called the Wig *service logic* (WSL). The specifics of WSL are dealt with later in this paper. One might argue in favor of other specification formalisms, e.g., Colored Petri Nets or Message Sequence Chart Diagrams. One of the reasons why we have chosen M2L-Str is

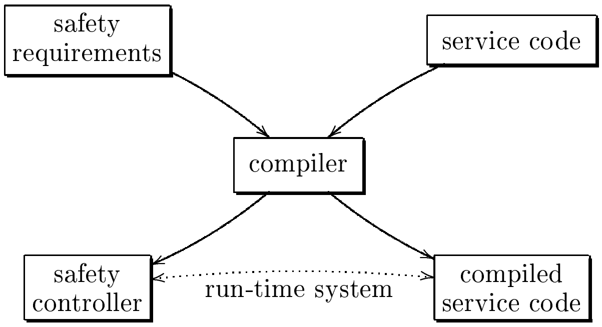


Fig. 2. Overview of the setup.

that we have a very well functioning tool available for doing computations in this logic [4]. Also, since we do not aim for traditional model checking, but rather for synthesizing controllers to be run on fast machines in slow networks, we are in the fortunate position to choose whatever logic provides the most succinct and intuitive syntax. All in all, M2L-Str is very powerful and yet just simple enough to actually allow calculations.

A formula ϕ in M2L-Str over the alphabet Σ will—when interpreted over a finite string w—either evaluate to true or to false and we shall write $w \models \phi$ or $w \not\models \phi$, respectively. The *language* associated with ϕ is

$$L(\phi) = \{\, w \in \Sigma^* \mid w \models \phi \,\}.$$

We shall denote by $pre(\phi)$ the prefix closure of $L(\phi)$.

We will use M2L-Str formulae as safety requirements as follows. Given a safety requirement ϕ we want to restrict the execution of the service S to allow only runs $\sigma = \sigma_0\sigma_1\sigma_2 \cdots \in \Sigma^\omega$ for which

$$\{\, \sigma_0 \ldots \sigma_{n-1} \in \Sigma^* \mid n \geq 0 \,\} \subseteq pre(\phi).$$

That is, we only allow a run σ if all its finite prefixes, $\epsilon, \sigma_0, \sigma_0\sigma_1, \sigma_0\sigma_1\sigma_2, \ldots$, are in the prefix closure of the language associated with ϕ.

Example 1. A safety requirement formula might look as follows.

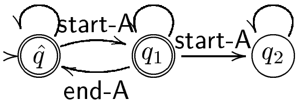
$$\begin{gathered}\forall\textbf{\textsf{time}}\ \mathsf{t,t''}\text{: } (\mathsf{t<t''} \land \mathsf{start\text{-}A(t)} \land \mathsf{start\text{-}A(t'')}) \\ \implies \exists\textbf{\textsf{time}}\ \mathsf{t'}\text{: } \mathsf{t<t'<t''} \land \mathsf{end\text{-}A(t')}.\end{gathered}$$

The formula will ensure that at most one session A thread will be allowed to execute at a time.

The above formula will occur automatically once per global variable X with start-A and end-A replaced by take-X and give-X respectively. This will ensure that only one session thread can update a given global variable at a time. One might argue that this could be implemented by simply using a semaphore for each global variable. True, but what we are dealing with here is just one specific safety requirement. Our technique can uniformly handle all safety requirements expressible in M2L-Str, i.e., errors are less likely to occur. Furthermore we will argue later that the technique handles specific requirements like critical regions as efficiently as if implemented directly, e.g., by means of a semaphore.

It has been known since the late sixties that M2L-Str characterizes regularity [8]. The Mona system provides an algorithm for translating M2L-Str formulae into minimal deterministic finite state automata (*mdfa*). Furthermore, regularity is preserved under prefix closure. Thus we have a method for producing from the safety requirements an *mdfa* that will function as our safety controller.

Example 2. The minimal safety controller corresponding to the requirement of Example 1 will thus be



with the convention that non-labeled transitions are implicitly labeled by Σ minus the labels that occur on other outgoing transitions from that state.

As can be seen, it is not possible to start a new session A thread if an A thread is already running.

4 Labeled Services with Safety Requirements

Given a service S with induced automaton A_S and a safety controller A_c we can quite precisely define the restricted behavior that we expect from the composite system. First a definition.

We define the *product automaton* $A_1 \times A_2$ of two automata A_1 and A_2, $A_i = (Q_i, \hat{q}_i, \Sigma, \rightarrow_i, F_i)$ to be

$$A_1 \times A_2 = (Q_1 \times Q_2, (\hat{q}_1, \hat{q}_2), \rightarrow, F_1 \times F_2),$$

where

$$(q_1, q_2) \xrightarrow{\sigma} (q_1', q_2') \text{ iff } q_i \xrightarrow{\sigma}_i q_i' \text{ for } i = 1, 2.$$

Thus the restricted behavior that we want our composite system to have is that of $A_S \times A_c$. For a product $A_1 \times A_2$ of two systems A_1 and A_2 we have $L(A) = L(A_1) \cap L(A_2)$. Thus the product of the service and the controller will allow—among the possible runs of the service—exactly those that also meet the safety requirements.

4.1 Implementing the Naïve Runtime System

The service does not constitute a finite state automaton. Therefore we cannot produce the full runtime system (the combined system consisting of both the service A_S and the controller A_c) by simply computing the product automaton at compile time. What we will do instead is to implement the implicit synchronization of the product automaton directly as part of the runtime system. The runtime system will then consist of three parts:

- a safety controller A_c,
- for each label $\sigma \in \Sigma$ a run-time queue $\mathrm{rtq}(\sigma)$, and
- the current session threads of the service.

All session threads and the controller run in parallel having the queues as shared resources.

- The code for the sessions will then be compiled such that each time a session thread wants to pass a label σ it pushes its session thread *id* onto $\mathrm{rtq}(\sigma)$ and then waits for permission to continue. When permission later is granted by the controller it will pass the label σ and continue its execution.
- The safety controller will be looping while doing the following. Check if any of the queues corresponding to the enabled transitions are non-empty. In case it finds a non-empty queue, say $\mathrm{rtq}(\sigma)$, it
  1. removes a session thread *id* from $\mathrm{rtq}(\sigma)$,
  2. changes its state corresponding to making the enabled σ-transition, and
  3. wakes up the session thread corresponding to the removed *id*.

This way the runtime system will behave as the product of the service and the controller. A diagram of a simple runtime system can be found in Fig. 3.

5 Improvements on the Runtime System

In the following section we will present two major improvements on the runtime system. Since the service part of the runtime system is very hard to reason about at compile time, our improvements will concentrate on the safety controller (which is just a finite automaton) and the shared queues. Both improvements are achieved by using the notion of distributed automata.

- The first improvement concerns the avoidance of the state explosion problem.
- The second improvement increases parallelism of the system by inferring independence information.

First let us define the notion of distributed automata. A *distributed alphabet* $\tilde{\Sigma} = (\Sigma_1, \ldots, \Sigma_K)$ with $K \geq 1$ is a finite collection of finite, non-empty alphabets. We will denote by Loc the set $\{1, \ldots, \mathrm{K}\}$ and by Σ the union of the not necessarily disjoint alphabets Σ_i. By $\mathrm{loc}(\sigma)$ we will denote the set of locations where σ occurs, i.e., $\mathrm{loc}(\sigma) = \{\, i \in \mathrm{Loc} \mid \sigma \in \Sigma_i \,\}$.

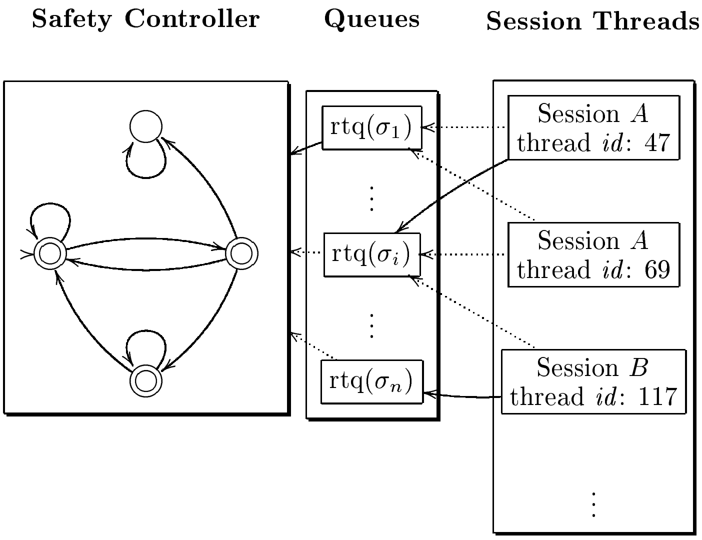


Fig. 3. Sketch of the runtime system.

A *distributed automaton* over $\tilde{\Sigma}$ is a structure $\mathcal{A} = (A_1, \ldots, A_K)$ consisting of finite state automata $A_1, \ldots, A_K$, with $A_i = (Q_i, \hat{q}_i, \Sigma_i, \rightarrow_i, F_i)$. The derived behavior of $\mathcal{A}$ will be as the behavior of the finite state automaton

$$A = (Q_1 \times \cdots \times Q_K, (\hat{q}_1, \ldots, \hat{q}_K), \Sigma, \rightarrow, F_1 \times \cdots \times F_K),$$

where $(q_1, \ldots, q_K) \xrightarrow{\sigma} (q'_1, \ldots, q'_K)$ if and only if
– when $i \in \mathrm{loc}(\sigma)$ then $q_i \xrightarrow{\sigma}_i q'_i$ and
– if $i \notin \mathrm{loc}(\sigma)$ then $q_i = q'_i$.

We will denote by $L(\mathcal{A})$ the language recognized by the distributed automaton $\mathcal{A}$ which is just the language recognized by A, i.e., $L(\mathcal{A}) = L(A)$.

Note that given a distributed automaton $\mathcal{A} = (A_1, \ldots, A_K)$ over $\tilde{\Sigma}$ we also have

$$L(\mathcal{A}) = \{\, w \in \Sigma^* \mid \forall i \in \mathrm{Loc} : (w|\Sigma_i) \in L(A_i) \,\},$$

where $w|\Sigma_i$ denotes the projection of the string w onto the ith alphabet Σ_i.

Note that for $K = 1$ the notion of a distributed automaton coincides with the simple notion of a finite state automaton. The kind of distributed automata that we will be using in this first part to reduce the state space will be automata over distributed alphabets where $\Sigma_1 = \cdots = \Sigma_K$, i.e., the derived behavior of the distributed automaton will simply be as the behavior of the product $A_1 \times \cdots \times A_K$. Later on—when we want to increase parallelism—we will consider general distributed alphabets where the sub-alphabets are not necessarily equal to each other.

5.1 The State Explosion Problem

In model checking, the state explosion problem occurs very often and there have been many attempts to avoid it, e.g., by means of a symbolic representation using BDDs [7,1]. As a colloquial remark one might mention that we actually already do use BDDs because of the way the Mona system represents its data.

Here we attack the state explosion problem by using distributed automata. The crucial observation that makes it possible to easily use distributed automata in this context is that safety requirements have the form of a big conjunction, i.e., it is a collection of requirements that all have to be satisfied. Also, we can build and use the symbolic representation of the products (the distributed automaton) all the way through when generating controllers. In model checking, though, one needs at some point to actually compute the product.

Let C be a set of safety constraints, i.e., a set of conjuncts. Then, given some partition of C into $C_1, \dots, C_K$, we can compile each of the smaller conjunctions $C_1, \dots, C_K$ separately into automata $A_1, \dots, A_K$. The corresponding distributed automaton $\mathcal{A} = (A_1, \dots, A_K)$ will then behave exactly as the automaton corresponding to the full set C of conjuncts because

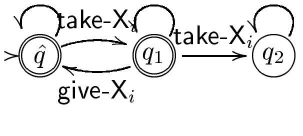
$$\begin{aligned} L(C) &= L(C_1 \wedge \cdots \wedge C_K) \\ &= L(C_1) \cap \cdots \cap L(C_K) \\ &= L(A_1) \cap \cdots \cap L(A_K) \\ &= L(A_1 \times \cdots \times A_K) \\ &= L(\mathcal{A}). \end{aligned}$$

We can thus use $\mathcal{A}$ as our safety controller instead. If the partition is chosen appropriately this will lead to a considerable reduction of the state space.

Example 3. Consider the example of having n global variables $X_1, \dots, X_n$. In order to obey the requirement of mutual exclusion on assignment to globals this will automatically generate the following list of safety constraints.

$$\begin{aligned} &\forall\textbf{time}\ \mathsf{t,t''}\text{: } (\mathsf{t<t''} \wedge \mathsf{take\text{-}X_1(t)} \wedge \mathsf{take\text{-}X_1(t'')}) \\ &\qquad \Longrightarrow\ \exists\textbf{time}\ \mathsf{t'}\text{: } \mathsf{t<t'<t''} \wedge \mathsf{give\text{-}X_1(t')}; \\ &\qquad\qquad\vdots \\ &\forall\textbf{time}\ \mathsf{t,t''}\text{: } (\mathsf{t<t''} \wedge \mathsf{take\text{-}X_{\mathit{n}}(t)} \wedge \mathsf{take\text{-}X_{\mathit{n}}(t'')}) \\ &\qquad \Longrightarrow\ \exists\textbf{time}\ \mathsf{t'}\text{: } \mathsf{t<t'<t''} \wedge \mathsf{give\text{-}X_{\mathit{n}}(t')}; \end{aligned}$$

The corresponding safety controller will have $2^n + 1$ states—it will look like the nth dimensional cube plus the "error state". If we use the distributed automaton approach and partition the safety requirements according to the semicolons then the controller will have only a linear ($3n$) number of states. We will have n copies of the following three state automaton.



5.2 Inference of Independence Information

It was shown in the previous section that using distributed automata reduces the state space in frequently occurring cases. In this section we will improve on the fact that we have a central component—the safety controller—which can potentially slow down the performance of the service, e.g., if many session threads are asking for permission to continue at the same time. This can—in many cases—be avoided by exploiting independence in the safety requirements.

We shall call a transition $q \xrightarrow{\sigma} q'$ *state preserving* if $q = q'$. A label σ is said to be *dead* if all σ-transitions are state preserving, i.e., if

$$\rightarrow \cap (Q \times \{\sigma\} \times Q) \subseteq \{\, q \xrightarrow{\sigma} q \mid q \in Q \,\}.$$

A σ-transition is *dead* if σ is dead.

In a distributed automaton $\mathcal{A} = (A_1, \ldots, A_K)$ a label or transition is said to be *locally dead* in A_i if it is dead in A_i. One can now make the following observation.

Proposition 1. *Given a distributed automaton $\mathcal{A} = (A_1, \ldots, A_K)$. Let $\mathcal{A}'$ be the distributed automaton $\mathcal{A}$ where locally dead labels and transitions have been removed. If $\Sigma' = \Sigma$ then $L(\mathcal{A}') = L(\mathcal{A})$.*

That is, if we do not remove the last occurrence of a label in $\mathcal{A}$ then removing locally dead labels and transitions is a language preserving operation.

The above proposition easily extends to a simple algorithm where we iterate through the automata $A_1, \ldots, A_K$ and for each $i \in \mathrm{Loc}$ remove all locally dead labels and transitions. Running this algorithm on a distributed automaton $\mathcal{A}$ will result in a new distributed automaton $\mathcal{A}'$ with exactly the same overall behavior, but with minimized requirements regarding synchronization between the different components.

Now—to improve performance even further—consider the undirected graph $G = (V, E)$ with nodes $V = \{A_1, \ldots, A_K\}$ and edges $E = \{\, (A_i, A_j) \mid \Sigma_i \cap \Sigma_j \neq \emptyset \,\}$. The connected components $C_1, \ldots, C_n$ of G can—since they are collections of finite automata—be considered as distributed automata.

The crucial observation is that—since they have completely disjoint alphabets—these distributed automata $C_1, \ldots, C_n$ can be run completely in parallel while still guaranteeing all safety requirements to be met. Furthermore, inside each of the distributed automata C_i we still have finite state automata that are loosely coupled and thus need very little internal synchronization as well.

A diagram of the runtime system with n global variables, $X_1, \ldots, X_n$, where we have both reduced the state space and exploited independence information can be found in Fig. 4.

As can be seen, we handle the case of critical regions just as efficiently as if we had implemented it using a semaphore, both with respect to space requirements and with respect to the degree of parallelism. Thus we do not lose anything by formulating simple safety requirements in this way. What we gain, however, is a uniform framework in which we can formulate all safety requirements.

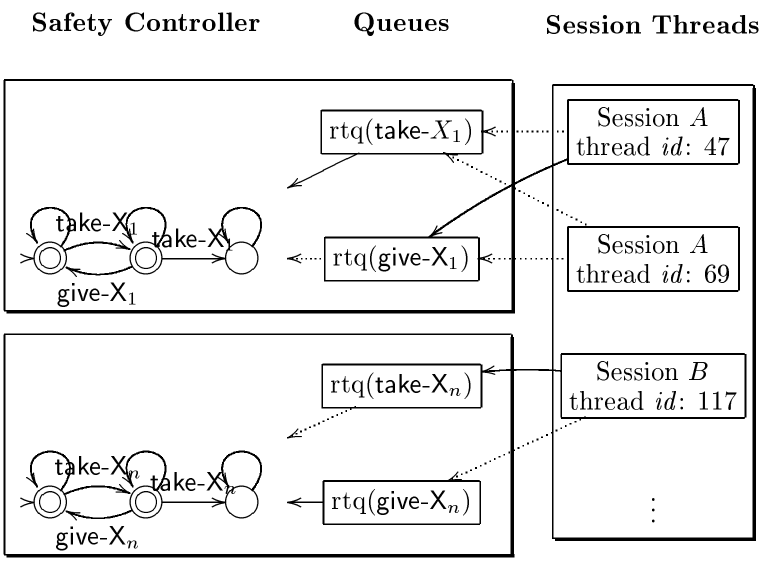
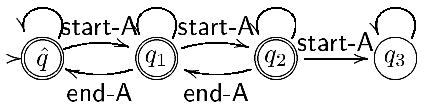


Fig. 4. Runtime system with independent distributed automata.

Inference of independence information via search for locally dead labels and transitions will thus in general lead to a substantial increase of parallelism in the controller. This increase of parallelism will improve on the overall performance of the runtime system, i.e., on the flow capacities of the provided service.

6 Beyond Regularity

In general one can have any number of session A threads. This can of course be constrained to any fixed maximal number by use of safety requirements, e.g., the controller



will guarantee that at any time there will be at most two session A threads.

But very often one has requirements like: I only want to enter this part of the code, e.g., pass label foo, if there are no session C threads—without putting a bound on the maximal number of session C threads. This cannot be expressed in M2L-Str. The positions at which there are no session C threads are those where the number of start-C and end-C labels occurring before that position are the same but the standard example of a *non*-regular language is $\{ a^n b^n \mid n \geq 0 \}$ thus the property is not regular and therefore cannot be expressed in M2L-Str.

In order to accommodate the need for these kind of requirements we will introduce the notion of a *counter*. We could declare the counter last-A as follows.

$$\textsf{\textbf{counter} last-A : \#start-A} - \textsf{\#end-A;}$$

This would have the effect that an extra label, last-A, would be added to the alphabet. The extra label would be passed implicitly every time the right-hand side of the counter declaration reaches zero, i.e., in this case every time the last session A thread has passed its end-A label. Thus, we produce a controller that only allows runs that both pass the safety requirements and furthermore have the property that last-A occurs exactly at positions following occurrences of end-A that results in a prefix that has the same number of start-A and end-A occurrences.

So in principle we want to make an intersection of a regular and a context free language which in general of course is non-regular. We still want to implement the controller as a finite state (distributed) automaton but this is no longer possible. However, if we equip the automaton with integer variables—one for each counter—then we will have enough machinery to recognize the intersection of the two languages. More specifically, consider the three step loop in the description of the controller in the naïve runtime system:

> The safety controller will be looping while doing the following. Check if any of the queues corresponding to the enabled transitions are non-empty. In case it finds a non-empty queue, say rtq(σ), it
> 1. removes a session thread *id* from rtq(σ),
> 2. changes its state corresponding to making the enabled σ-transition, and
> 3. wakes up the session thread corresponding to the removed *id*.

Apart from the fact that there are now several of these controllers each having a subset of labels to take care of, we must add the following conditional as a fourth step.

> If σ occurs on the right-hand side of a counter declaration, $\textsf{cnt}_i$, then increment or decrement the variable associated with that counter. If it reaches zero then change the state corresponding to taking the $\textsf{cnt}_i$-transition.

By using the counter last-A we thus

- add the label last-A and
- in the controller we intersect with the language where last-A occurs after the termination of the last session *A* thread.

Therefore, one can now write a safety requirement ensuring that there are no active *A* sessions. E.g., the predicate

$$\begin{aligned}\textsf{zero-A(t'')} \equiv \exists\textbf{time}\ \textsf{t: t}\leq\textsf{t''} \wedge (\textsf{t=0} \vee \textsf{last-A(t)}) \wedge \\ \forall\textbf{time}\ \textsf{t': t}\leq\textsf{t'}\leq\textsf{t''} \implies \neg\textsf{start-A(t')}\end{aligned}$$

will only evaluate to true at positions where there are no active *A* sessions.

Example 4. We can now formulate the (non-regular) requirement from the beginning of this section, "I only want to pass label foo if there are no active *C* sessions", as a safety requirement:

$$\forall\textbf{time}\ \textsf{t: foo(t)} \implies \textsf{zero-C(t)}.$$

7 WSL

The Wig service logic (WSL) is a high-level notation built on top of M2L-Str suitable for writing safety requirements for Wig service code implementing an interactive web service. It inherits from M2L-Str the usual universal and existential quantifications over both first order variables (ranging over instances of discrete time) and monadic second order variables (ranging over sets of instances of time). Also, it has standard boolean connectives like negation, conjunction, disjunction, implication, etc., as well as operations on first order terms, e.g., given an instance of time t one can point out its successor (t+1), operations on second order terms, e.g., taking the union of two sets, plus of course the basic formula that tests membership of a position in a position set. Furthermore WSL provides basic formulae to test whether a label LL is passed at time t: LL(t).

Example 5. Consider some Wig code with a critical region that needs exclusive access to, say, a global resource. The way one makes the region critical is by first adding labels around it.

```
... code ...
label begin-crt-region;
... critical region ...
label end-crt-region;
... more code ...
```

Then, in order to make the code between these labels act as a critical region the following safety constraint is added to the set of requirements.

$$
\begin{aligned}
&\forall\textbf{time}\ \mathsf{beg_1},\ \mathsf{beg_2}:\\
&\qquad((\mathsf{beg_1}<\mathsf{beg_2}) \land \mathsf{begin\text{-}crt\text{-}region}(\mathsf{beg_1}) \land \mathsf{begin\text{-}crt\text{-}region}(\mathsf{beg_2}))\\
&\qquad\Longrightarrow\ \exists\textbf{time}\ \mathsf{end_1}\text{: } (\mathsf{beg_1}<\mathsf{end_1}<\mathsf{beg_2}) \land \mathsf{end\text{-}crt\text{-}region}(\mathsf{end_1}).
\end{aligned}
$$

Example 6. Another thing that WSL is suitable for is formulating requirements regarding priority. We are given some service S with sessions Reading and Writing (See Fig. 5). Reading threads read data from a Database, display the data in a proper way, read some more data, display it, and so forth. Writing threads can only be started by the service administrator. Furthermore, at most one Writing session thread is allowed at a time. This last condition can easily be satisfied, e.g., by the constraint

$$\forall\textbf{time}\ \mathsf{t}\text{: } \mathsf{start\text{-}Writing(t)} \implies (\mathsf{t{=}0} \lor \mathsf{zero\text{-}Writing(t\text{-}1)}),$$

where zero-Writing and later zero-Reading are defined as the zero-A predicate from the previous section.

Of course Writing must not be started unless there are no active Reading threads and vice versa. This can also be formulated in a straightforward way using, e.g., the constraint

$$\forall\textbf{time}\ \mathsf{t}\text{: } \mathsf{zero\text{-}Writing(t)} \lor \mathsf{zero\text{-}Reading(t)}.$$

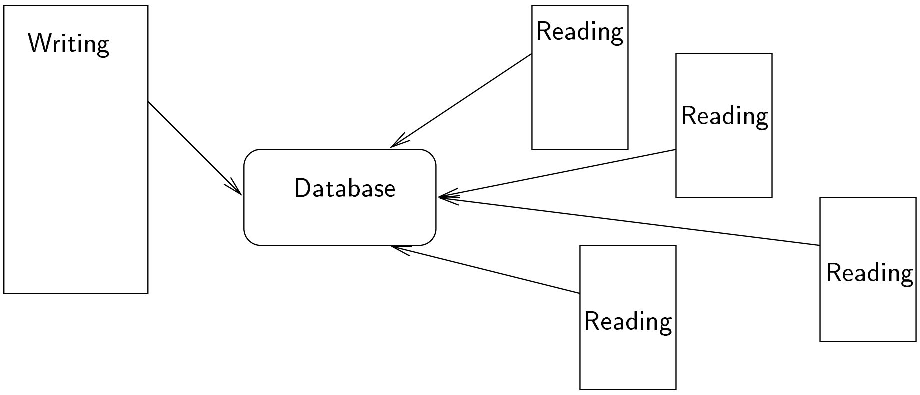


Fig. 5. A Writing thread and several Reading threads accessing a Database.

But what if it is of great importance that the administrator gets access to the database as soon as possible? E.g., if the database contains prices of products that a company sells and corrections have to be made to these prices. Thus, we want to give Writing priority over Reading. This can be managed in the following way (excluding the last WSL-formula above). We assume that the critical regions of Writing, i.e., those that access the database, are surrounded by labels start-Crt and end-Crt.

- Then, in order to make sure that as soon as the Writing thread has started, no more Reading threads will start, we add the constraint

  $$\forall \textbf{time}\ \mathsf{t}\colon \mathsf{start\text{-}Reading(t)} \implies \mathsf{zero\text{-}Writing(t)}.$$

- Of course, we should still make sure that we have mutual exclusion. This is done by adding the constraint

  $$\forall \textbf{time}\ \mathsf{t}\colon \mathsf{start\text{-}Crt(t)} \implies \mathsf{zero\text{-}Reading(t)}.$$

  Thus, we make sure that we do not enter any critical region in Writing unless the last Reading thread has ended its execution.

Using the above approach we satisfy the crucial property of mutual exclusion. Furthermore, we impose restrictions that will make sure the administrator gets priority over ordinary users. The corresponding automaton, however, is rather complex, see Fig. 6. Thus, producing it by hand is cumbersome and error prone.

Furthermore, our approach is modular in the sense that requirements can be added and deleted at will without changing the existing requirements. In the automata world small changes can result in extensive changes of the automaton. Solutions using semaphores lack in a similar fashion the property of being modular.

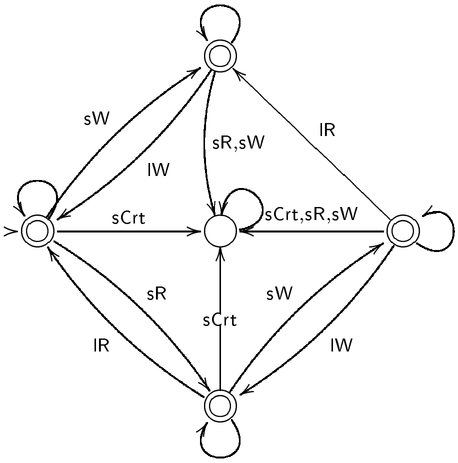


Fig. 6. Automaton corresponding to the Writing/Reading setup.

8 Related Work

Mawl has already been mentioned as an example of a domain-specific language for describing sequential transaction-oriented Web applications. The techniques available for doing synchronization when dealing with concurrency in Mawl is limited to working with critical regions, though. Our work on service logic extends this to working with arbitrary safety constraints.

In general, there are increasingly many systems for doing web-programming, e.g., [6, 5, 3], but so far none of seem of them seem to support proper handling of safety requirements.

The area of control theory is of course huge. We are only dealing with control of discrete systems, though. Ramadge and Wonham give in [9] a good survey on "The Control of Discrete Event Systems". Many of the notions presented here are similar to those of [9].

Distributed automata are simply a special case of the product automata of [10]. They again are a special case of the non-cellular asynchronous automata of [11].

9 Future Work

Writing Wig services results in generation of highly concurrent code which needs lots of synchronization. The implementation of the ideas presented here is in preparation and once we can do experiments writing and using safety constraints, evaluation of the usefulness of our technique can be taken further. The important question is of course: how much do these ideas improve on the quality of the services?

When dealing with inference of independence information, the important part is that of choosing the right partition of the safety requirement. We plan to do static analyses on the safety requirements and combine the achieved information with appropriate heuristics to obtain hopefully good results. Intuitively, the formulae of the safety requirements implicitly say something about which requirements are closely related and which are not.

The idea of having a central controller is proving useful for other aspects of web services. We plan to include support for various kinds of event handling and database locking. Also, the controller may drive an automatically generated service monitor, allow maintenance and performance statistics.

10 Acknowledgments

The authors thank Claus Rasmussen Brabrand and the anonymous referees for useful comments.

References

1. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
2. K. Cox, T. Ball, and J. C. Ramming. Lunchbot: A tale of two ways to program web services. Available from http://www.cs.utexas.edu/users/cpg/mawl/doc/lunchbot.ps.gz, April 1996.
3. Brian J. Fox. Meta-html: A dynamic programming language for www applications. http://www.metahtml.com/documentation/manifesto.html.
4. J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019*, 1996.
5. The document *is* the application. Web Site. http://www.htmlscript.com/.
6. D. A. Ladd and J. C. Ramming. Programming the web: An application-oriented language for hypermedia services. In *4th Intl. World Wide Web Conference*, 1995.
7. K. L. McMillan. *Symbolic Model Checking*. Kluver Academic Publishers, 1993.
8. M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. Amer. Math. Soc.*, 141:1–35, 1969.
9. Peter J. G. Ramadge and W. Murray Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, January 1989.
10. P. S. Thiagarajan. PTL over product state spaces. Technical Report TCS-95-4, School of Mathematics, SPIC Science Foundation, 1995.
11. W. Zielonka. *The Book of Traces*, chapter Asynchronous Automata, pages 205–248. World Scientific Publishing, 1995.