Applying Static Analysis to Software Architectures

Gleb Naumovich, George S. Avrunin, Lori A. Clarke and Leon J. Osterweil

email: {naumovic|avrunin|clarke|ljo}@cs.umass.edu

Laboratory for Advanced Software Engineering Research
Computer Science Department
University of Massachusetts
Amherst, Massachusetts 01003

Abstract

In this paper we demonstrate how static concurrency analysis techniques can be used to verify application-specific properties of an architectural description. Specifically, we use two concurrency analysis tools, INCA, a flow equation based tool, and FLAVERS, a data flow analysis based tool, to detect errors or prove properties of a Wright architectural description of the gas station problem. Although both these tools are research prototypes, they illustrate the potential of static analysis for verifying that architectural descriptions adhere to important properties, for detecting problems early in the lifecycle, and for helping developers understand the changes that need to be made to satisfy the properties being analyzed.

1 Introduction

With the advent of improved network technology, distributed systems are becoming increasingly common. Such systems are more difficult to reason about than sequential systems because of their inherent nondeterminism. In recognition of this, software architecture research is attempting to define architectural languages to help developers describe distributed system designs. These high-level descriptions allow developers to focus on structural, high-level design issues before lower level details are addressed, thereby helping to discover areas of high risk and to address these risks as early in the lifecycle as possible. To be truly beneficial, developers should be given tools to help them reason about their architectural descriptions, to help them discover problems as early as possible, and to help them verify that desired properties would indeed be maintained by these designs as well as by any systems correctly derived from these designs. It has been demonstrated that detecting errors early in the lifecycle greatly reduces the cost of fixing those errors. Architectural description languages combined with appropriate analysis tools could therefore be an important means for reducing costs and improving reliability.

A number of architectural description languages have been proposed, such as Wright [2], Rapide [13], Darwin [14, 15], and UniCon [19]. There has also been some work on validating aspects of architectural designs. Using architectures specified in UniCon, for instance, developers can estimate local timing information and use those estimates to check time-dependent properties with the RMA real-time analysis tool [12]. Another approach is to use model-theoretic proof techniques use model-theoretic proof techniques to verify conformance of elaborated architecture descriptions to higher-level architectural designs [14, 18]. Developers using the Rapide architectural description language can simulate executions of the system and verify that the traces of those executions conform to high-level specifications of the desired behavior [13]. Although one would expect the number of traces through an architectural description to be much less than the number of possible executions in the corresponding software system, for most interesting systems there are far too many such traces to explore them all. Thus, this is basically a sampling technique, and while it increases confidence in the design, it does not verify that all executions conform to the specifications. Another validation approach that has been explored is the use of static analysis techniques to verify general properties
of architectural descriptions. When successful, this type of analysis does verify that all possible executions conform to the specification. Allen and Garlan [1] use the static analysis tool FDR [7] to prove freedom from deadlock as well as compatibility between the components and connectors in an architectural description. These are general properties that are desirable for all architectural descriptions.

The primary goal of this work is to investigate the applicability of existing static analysis techniques for verifying application-specific properties of architectural designs. We investigate one example architecture, a Wright description of the gas station problem, and illustrate the kinds of properties that can be verified and the kinds of errors that can be found early in the lifecycle. Two versions of a Wright architectural specification of the gas station example were graciously provided to us by David Garlan. We applied two static analysis tools: INCA, which is based on flow equations, and FLAVERS, which is based on data flow analysis. Both of these tools are research prototypes; they illustrate the potential of static analysis for verifying that architectural descriptions adhere to important properties, for detecting problems early in the lifecycle, and for helping developers understand the changes that need to be made to satisfy the properties being analyzed.

The next section gives a high-level overview of the two static analysis tools used in this case study. Section 3 gives a brief description of the gas station problem and the Wright specification of the problem. Section 4 introduces the properties we selected to prove about this architecture and describes the analysis process and the results of that process. Section 5 summarizes the overall results, describes the benefits of this approach, and points out some interesting directions for future research.

2 Tools Used

A number of automated static concurrency analysis techniques have been proposed. They span such approaches as reachability analysis (e.g. [11, 20, 8]), symbolic model checking [4, 17], flow equations [5], and data flow analysis [6, 16]. The goal of this work is to demonstrate the applicability of static analysis techniques to architecture descriptions but not, at least at this point in time, to determine which approach might be best. Thus, we selected two different static analysis tools, based on fundamentally different approaches but on which we have considerable expertise. One tool, INCA [5], is based on flow equations, and the other, FLAVERS [6], is based on data flow analysis. Both these tools can be used to check whether all executions of a concurrent system satisfy a property, such as the mutually exclusive use of some resource. Although these tools use different approaches, they both are conservative in that if they determine that a property holds, it is guaranteed to hold for all executions. When a property fails to hold, however, this may be because the system does indeed violate the property or it may be because the analysis, in order to assure conservativeness and improve efficiency, has over-approximated the executable behavior of the system. Thus, when a property fails to hold, the results are inconclusive and usually require further investigation. A brief description of each of these tools is given here.

**Inequality Necessary Conditions Analysis (INCA)** derives a set of necessary conditions for the existence of an execution violating the property. In INCA, the sequential processes making up the concurrent system are translated into finite state automata (FSAs) from which necessary conditions, expressed as linear inequalities on the occurrences of transitions in these automata, are derived. These inequalities reflect certain kinds of compatibility conditions among the executions of the individual processes that must be satisfied in an execution of the full program. The violation of the property is also expressed as inequalities in terms of occurrences of the FSA transitions. The consistency of the resulting system of linear inequalities is checked using standard integer linear programming (ILP) techniques. This approach is inherently compositional, in the sense that the inequalities are generated from the automata corresponding to the individual processes, rather than from a single automaton representing the full concurrent system. Thus, INCA avoids considering the state space of the full system. The size of the system of inequalities is essentially linear in the number of processes in the system. Furthermore, the use of properly chosen cost functions in solving the ILP problems can guide the search for a solution. ILP is itself an NP-hard problem in general, and the standard techniques for solving ILP problems (branch-and-bound methods) are potentially exponential. In practice, however,
the ILP problems generated from concurrent systems have large, totally unimodular subproblems and seem particularly easy to solve. Experience suggests that the time to solve these problems grows approximately quadratically with the size of the system of inequalities (and thus with the number of processes in the system).

The Flow Analysis for VERifying Software (FLAVERS) static analysis tool employs data flow analysis to verify that a model of the system must always be consistent with a property, perhaps restricted by a set of additional constraints. In FLAVERS, the control flow graph representation of each sequential process, annotated with events of interest, is composed into a task flow graph, which explicitly represents the communications among the distributed processes as well as the interleavings of events among those processes. The properties to be checked are translated into a finite state automaton, where the transitions are annotated with the appropriate events of interest. Using a data flow analysis algorithm that is $O(N^2S)$, where $N$ is the node size of the task flow graph and $S$ is the state size of the automaton, FLAVERS determines whether the language of the system is accepted by the language of the automaton. If at the terminal node of the flow graph all event sequences are in the language of the property, we know that the property holds on all executions of the system. When some event sequences are in the language of the property and some are not, the results of the analysis are inconclusive, since it has to be determined whether the event sequences that violate the property happen on any real executions of the system. FLAVERS offers a means to deal with inconclusive results by allowing the analyst to add additional constraints, in the form of finite state automata, which limit the behaviors represented by the task flow graph. For example, a constraint can model the behavior of a single variable in the system. This additional information about the system restricts the data propagation through the flow graph during the analysis, thereby improving the accuracy of the analysis.

INCA and FLAVERS are based on very different analysis techniques, although both avoid enumerating the total state space of a distributed system. In addition, both techniques have been used to prove a wide range of properties of distributed systems. Because of this and our expertise with these tools, we chose them for our initial exploration of analyzing application-specific properties of architectures.

3 Architectural Specification of the Gas Station Example

The Gas Station system [9] models a self-serve gas station. This example has been widely studied by the static analysis research community. It has also been used in the software architecture community, and was the example provided to us by Garlan. In the general case, this system consists of $n$ customers who come to a gas station to obtain gas for their vehicles, $m$ cashiers who sell the gas, and $p$ pumps that discharge the gas. The customers pay the cashiers (and get change in some versions), who order the pumps to discharge gas. We consider a specific instance of this system, with two customers, one cashier, and one pump. Garlan gave us Wright specifications for two versions of this system.

Wright formally describes architectures as collections of components, which represent computation units in the system, and connectors, which represent the means of information exchange among the components. Each component and connector is augmented with specifications that permit one to characterize precisely the abstract behavior of the components and their interactions. For a component the specification consists of a number of ports, and a computation. Each port represents an interaction in which the component may participate. In other words, a port partially describes the interface of the component, taking the point of view of the connector or connectors that communicate with this component through this port. The computation describes the internal functionality of the component. A connector is represented by a set of roles specifying the interface of this connector and the glue that specifies how the interaction actually takes place. A system specification is composed of a set of component and connector type definitions, as described above, a set of instantiations of specific objects of these types, and attachments. Attachments specify which components are linked to which connectors. Wright uses CSP [10] to describe the behavior of roles ports, computations, and glues.

Figure 1 shows the Wright specification for the first version of the Gas Station. This architecture de-
Component Customer
  Port Pay = pay!x → Pay
  Port Gas = take → pump?x → Gas
  Computation = Pay,pay!x → Gas,take → Gas,pump?x → Computation

Component Cashier
  Port Customer1 = pay?x → Customer1
  Port Customer2 = pay?x → Customer2
  Port Topump = pump?x → Topump
  Computation = Customer1,pay?x → Topump,pump?x → Computation
  □ Customer2,pay?x → Topump,pump?x → Computation

Component Pump
  Port Oil1 = take → pump?x → Oil1
  Port Oil2 = take → pump?x → Oil2
  Port Fromcashier = pump?x → Fromcashier
  Computation = Fromcashier,pump?x → (Oil1,take → Oil1,pump?x → Computation)
  □ (Oil2,take → Oil2,pump?x → Computation)

Connector Customer_Cashier
  Role Givemoney = pay!x → Givemoney
  Role Getmoney = pay?x → Getmoney
  Glue = Givemoney,pay!x → Getmoney,pay!x → Glue

Connector Customer_Pump
  Role Getoil = take → pump?x → Getoil
  Role Giveoil = take → pump?x → Giveoil
  Glue = Getoil,take → Giveoil,take → Giveoil,pump?x → Giveoil,pump?x → Glue

Connector Cashier_Pump
  Role Tell = pump?x → Tell
  Role Know = pump?x → Know
  Glue = Tell,pump?x → Know,pump?x → Glue

Instances
  Customer1: Customer
  Customer2: Customer
  cashier: Cashier
  pump: Pump
  Customer1_cashier: Customer_Cashier
  Customer2_cashier: Customer_Cashier
  Customer1_pump: Customer_Pump
  Customer2_pump: Customer_Pump
  cashier_pump: Cashier_Pump

Attachments
  Customer1_Pay as Customer1_cashier,Givemoney
  Customer1_Gas as Customer1_pump,Getoil
  Customer2_Pay as Customer2_cashier,Givemoney
  Customer2_Gas as Customer2_pump,Getoil
  cashier_Customer1 as Customer1_cashier,Getmoney
  cashier_Customer2 as Customer2_cashier,Getmoney
  cashier_Topump as cashier_pump,Tell
  pump_Fromcashier as cashier_pump,Know
  pump_Oil1 as Customer1_pump,Giveoil
  pump_Oil2 as Customer2_pump,Giveoil

Figure 1: Wright First Version of the Specification of the Gas Station
Component Customer

\[
\text{Port Pay} = \frac{\text{pay}!x}{\text{Pay}} \\
\text{Port Gas} = \frac{\text{pump}?x}{\text{Gas}} \\
\text{Computation} = \text{Pay}.\text{pay}!x \rightarrow \text{Gas}.\text{pump}?x \rightarrow \text{Computation}
\]

Component Pump

\[
\text{Port Oil}_1 = \frac{\text{pump}?x}{\text{Oil}_1} \\
\text{Port Oil}_2 = \frac{\text{pump}?x}{\text{Oil}_2} \\
\text{Port Fromcashier} = \frac{\text{pump}?x}{\text{Fromcashier}} \\
\text{Computation} = \text{Fromcashier}.\text{pump}_1?x \\
\(\text{Oil}_1.\text{pump}?x \rightarrow \text{Computation}) \\
\(\text{□ Fromcashier}.\text{pump}_2?x \rightarrow \text{Oil}_2.\text{pump}?x \rightarrow \text{Computation})
\]

Component Cashier

\[
\text{Port Customer}_1 = \text{pay}?x \rightarrow \text{Customer}_1 \\
\text{Port Customer}_2 = \text{pay}?x \rightarrow \text{Customer}_2 \\
\text{Port Topump} = \frac{\text{pump}?x}{\text{Topump}} \sqcap \frac{\text{pump}?x}{\text{Topump}} \\
\text{Computation} = \text{Customer}_1.\text{pay}?x \rightarrow \text{Topump}.\text{pump}_1?x \rightarrow \text{Computation} \\
\(\text{□ Customer}_2.\text{pay}?x \rightarrow \text{Topump}.\text{pump}_2?x \rightarrow \text{Computation})
\]

Figure 2: Wright Components of the Second Version of the Architecture

scribes three types of components and three types of connectors for communications between the customers and the cashier, the cashier and the pump, and the customers and the pump. The concrete instantiation of this architecture contains four components, Customer1, Customer2, Cashier, and Pump and five connectors, Customer1_cashier, Customer2_cashier, Cashier_pump, Customer1_pump, and Customer2_pump. As illustrated in Figure 1, each Customer component has two ports, where Pay specifies the behavior of the Customer as viewed by the Customer_cashier connector, and Gas specifies the behavior as viewed by the Customer_pump connector. The behavior of the Gas port consists of repeatedly taking the hose (take event) and pumping gas (pump?x event). The computation part of Customer specifies that a Customer does the following sequence of actions repeatedly: pay for gas, take the hose, obtain gas from the pump.

In this architecture, the customers repeatedly pay the cashier, then take the hose, and then wait for gas. The cashier, upon receiving a payment, turns the pump on. After a customer takes the hose and the pump receives authorization from the cashier, the pump then discharges the amount of gas, specified by the cashier, to the customer.

This version of the Gas Station example is known to have a critical race. Specifically, it is possible for Customer1 to pay before Customer2 pays but for Customer2 to take the hose before Customer1, thus getting the amount of gas purchased by Customer1.

The second version of the Gas Station removes this race by combining taking the hose and pumping the gas into a single action and by having the cashier tell the pump which customer should get gas. This means that, instead of paying and actively requesting gas by taking the hose, the customers now must pay and wait until the pump contacts them by sending gas. Figure 2 shows the second version of the specification for Customer, Pump, and Cashier components only, since changes to the connectors are trivial.

4 Checking Properties of the Gas Station Architecture

The existing versions of INCA and FLAVERS do not accept Wright specifications as input. While it should be relatively straightforward to build front-ends for both tools that would construct the appropriate internal representations directly from Wright, this seemed inappropriate for the initial exploration we had
task body Customer1 is
  cash : AMOUNT;
begin
  loop
    cash := SomeAmount;
    Customer1_cashier.getmoney_pay ( cash );
    Customer1_pump.getoil_take;
    accept gas_pump ( gas_amount : in AMOUNT );
  end loop;
end Customer1;

Figure 3: Ada Translation of the Customer Specification

in mind. Both tools accept Ada code as input, so we manually translated the Wright specifications into Ada in order to apply the tools. The close relationship between the concurrency constructs in CSP and Ada made this translation fairly easy. Each component and connector instantiation of the architecture is represented by an Ada task. The "$!" and "$?" operations of CSP naturally correspond to Ada rendezvous. The non-deterministic and deterministic CSP choice operators are modeled with the Ada select statement.

Figure 3 gives the Ada code for the Customer1 component for the first Wright specification. The assignment statement sets the variable cash to the value of a function whose body is not specified; the analysis tools treat this as a nondeterministic assignment. After choosing an amount of gas with this assignment, the Customer1 task calls the getmoney_pay entry of the the Customer1_cashier task with the parameter cash. This rendezvous corresponds to the pay!x event. The Customer1 task then calls the getoil_take entry of the Customer1_pump task, and then accepts a call at its own gas_pump entry. The complete Ada code we used for the various versions of the gas station is given in an appendix.

Our goal was to investigate whether existing static concurrency analysis tools could be usefully applied to check application-specific properties of architectural descriptions. Since the gas station is relatively simple, however, we focused on properties that reflect high-level requirements for a self-service gas station. Since we do not have any "official" requirements documents for the gas station, we chose a small number of properties that seemed to us to reflect reasonable requirements. Our goal was simply to explore the applicability of the static analysis tools to architectures; we make no claim that these are the most important or significant requirements.

In the remainder of this section, we show how INCA and FLAVERS were used to check several properties of the gas station architectures, identifying certain faults and verifying that modifications to the architectures corrected these faults.

4.1 The Critical Race to the Pump

As mentioned above, the first Wright specification has a critical race, in which one customer pays for gas and the second customer then pays and takes the pump before the first customer gets gas. In this case, the second customer gets the gas paid for by the first customer. The first requirement we considered was that customers get gas in the order in which they pay. We wanted to know whether INCA and FLAVERS could detect the violation of this property in the first Wright version, and whether they could show that the property holds in the second version.

We begin with the first version. The property we want to check is stated in terms of customers paying and getting gas. For the analysis, we must identify locations in the code that correspond to these events. We identified a customer paying with the corresponding rendezvous between the connector task from that customer to the cashier and the cashier task, and the customer getting gas with the rendezvous between the pump task and the connector task from the pump to the customer.

The INCA approach is to produce necessary conditions for an execution of the system that violates the property. We must therefore express a violation of the property as an INCA query. By symmetry, it is
\[
\text{(defquery "race" "nofair"
(omega-star-less
(\text{sequence}
\text{\begin{itemize}
\item \text{initial t : open t}
\item \text{ends-with } (rend "customer1\_cashier;cashier.\_customer1\_pay")
\end{itemize}})
\text{interval}
\text{\begin{itemize}
\item \text{ends-with } (rend "pump;customer2\_pump.\_getoil")
\item \text{require } (rend "customer2\_cashier;cashier.\_customer2\_pay")
\item \text{forbid } (rend "pump;customer1\_pump.\_getoil")
\end{itemize}})
\text{))))}
\text{)})
\]

Figure 4: INCA Query: Customers Get Gas in the Order They Pay.

enough to ask for an execution in which Customer2 pays and gets gas while Customer1 has paid but not yet gotten gas. So we wrote a query describing an execution in which a rendezvous between Customer1\_cashier and Cashier occurs, followed by a rendezvous between Customer2\_cashier and Cashier and a rendezvous between Pump and Pump\_Customer2 before the next rendezvous between Pump and Pump\_Customer1.

The INCA query we used is shown in Figure 4. This specifies a segment of an execution divided into two intervals. The first interval runs from the beginning of the execution (specified by the :initial keyword) and ends with some rendezvous between Customer1\_cashier and Cashier at the customer1\_pay entry (specified by the :ends-with keyword and the rend function). This interval is followed immediately by a second one that ends with a rendezvous between Pump and Customer2\_pump at the getoil entry of Customer2\_pump. The second interval contains a rendezvous between Customer2\_cashier and Cashier at the customer2\_pay entry (specified by the :require keyword) and does not contain any rendezvous between Pump and Customer1\_pump at the getoil entry (specified by the :forbid keyword).

From the Ada code corresponding to the first Wright specification and this query, INCA generated a system of inequalities. In this case, the system of inequalities had an integer solution, and INCA gave us the behavior of each task corresponding to that solution. From these task behaviors, it is straightforward to construct an execution in which the desired property is violated. To check this property for the second Wright specification, it was necessary to use two queries. (This is due to a technical reason involving certain cycles in the FSAs.) The first query checked that the cashier notifies the pump in the same order as customers pay, and the second query checked that the pump gives gas to the customers in the same order as it is notified by the cashier. The corresponding systems of inequalities were inconsistent, verifying that customers always get gas in the order that they pay with this second architecture.

The FLAVERS analysis is similar. For a FLAVERS analysis, the events of interest are indicated by annotating the Ada code. In this case, we used automatically generated annotations on the accept statements. For example, the “accept oil\_pump” statement in the Customer1 task was annotated with the event customer1\_oil\_pump. We then gave FLAVERS a property specification, in the form of a quantified regular expression (QRE), asking whether any execution could generate the sequence of events corresponding to a violation of the property. The QRE we used is shown in Figure 5. It consists of the alphabet, quantifier, and regular expression. The alphabet of the QRE appears in braces and lists all events used for the specification of the property. The alphabet is followed by the “none” quantifier instructing FLAVERS to attempt to verify the property that no execution leads to a sequence of the events in the alphabet that lies in the language of the regular expression that follows. In the regular expression, the period stands for the disjunction of all symbols, the notation [\text{-e}] stands for the disjunction of all symbols in the alphabet other than e, and the semicolon is the concatenation operator. The language of the regular expression thus consists of all strings over the alphabet in which a cashiers\_customer1\_pay occurs, followed by a cashiers\_customer2\_pay and a customer2\_pump\_getoil before a customer1\_pump\_getoil occurs.

For the first Wright specification, FLAVERS produces an execution in which the property is violated. For the second specification, FLAVERS verifies that the property holds for all executions.

Thus, both tools were able to detect the fault in the first version of the architecture, show how it occurs,
and verify that a modification to the architecture corrects the fault. The remaining properties were checked on this modified version.

4.2 No Free Gas

We next checked the requirement that no customer receive gas without paying for it. This amounts to checking that, in every execution and for each customer, the events of paying for gas and receiving gas strictly alternate, with paying for gas coming first. By symmetry again, it is sufficient to check this for Customer1. We used the same rendezvous corresponding to the events of the customer paying and getting gas as in the previous section.

Using INCA, the standard way to show two events alternate is to use two queries. In this case, the first query describes a prefix of an execution in which the number of times the customer has paid for gas exceeds the number of times it has received gas by at least two. The second query describes a prefix of an execution in which the number of times the customer has received gas is greater than the number of times the customer has paid for gas. (All INCA queries and FLAVERS QREs are shown in an appendix.) INCA reported that the necessary conditions for the existence of such executions were inconsistent. This means that, in every prefix of an execution, the number of times the customer has paid for gas is either equal to the number of times it has received gas or is one greater than the number of times the customer has received gas, showing that the events of paying for gas and receiving it strictly alternate, with paying for gas occurring first.

For FLAVERS, we used a QRE with the same alphabet as the one in Figure 5 and a regular expression requiring the two events to alternate appropriately. FLAVERS verified that the property holds on all executions.

4.3 Customers Get the Right Amount of Gas

We also checked whether a customer receives the amount of gas that he or she paid for. To facilitate the analysis, we allowed only two amounts (the type AMOUNT in our Ada programs had two values, 1 and 2). We then checked whether it was possible for a customer to pay for one amount of gas and then receive the other amount. By symmetry, it is sufficient to check only for one of the customers paying for one unit of gas and receiving two units.

Our INCA query asked for a prefix of an execution in which the first interval ends with a rendezvous with parameter 1 between Customer1_cashier and Cashier at the customer1_pay entry (the event where the customer pays for one unit of gas) and the second interval ends with a rendezvous with parameter 2 between Pump and Customer1_pump at the getoil entry (the event where the customer receives two units of gas). The second interval was forbidden to contain a rendezvous with parameter 1 between Pump and Customer1_pump at the getoil entry (the event where the customer receives the single unit of gas that was
paid for). INCA reported that the system of inequalities it generated was inconsistent, so no such execution could exist. This showed that customers never get the wrong amount of gas.

FLAVERS required additional event annotations to capture the numeric values of parameters that specify amounts of money and gas. Currently these annotations are manually added to the source code of the system under analysis in the form of comments. The QRE for this property specified that on no execution should it be possible that the event of \texttt{Cashier} receiving \texttt{1} at its \texttt{customer1.pay} entry is followed by the event of \texttt{Pump} giving \texttt{2} to the \texttt{getoil} entry of the \texttt{Customer1.pump} connector before \texttt{Pump} gives \texttt{1} to \texttt{Customer1.pump}. FLAVERS verified the property.

### 4.4 Another Race Condition

In checking the first two properties described earlier, we identified the event of a customer paying for gas with the \texttt{pay?x} action on the cashier's customer port (or, in the Ada code, with the corresponding rendezvous between the connector between the customer and cashier and the cashier task). Similarly, we identified the event of a customer receiving gas with the \texttt{pump?x} action on the pump's oil port (or with the corresponding rendezvous between the pump and the connector between the pump and customer). Viewing events as actions taken by components, we have here taken the viewpoint of the cashier and pump components about when a customer pays or receives gas. But we could just as well take the viewpoint of the customer component. In that case, we would identify the customer paying with the \texttt{pay?x} action on the customer's pay port and receiving gas with the \texttt{pump?x} action on the customer's gas port. The Ada rendezvous corresponding to the first action involves the customer and the \texttt{Customer.cashier} connector; the rendezvous corresponding to the second action involves the \texttt{Customer.pump} connector. In essence, we checked whether the pump “believes” customers get gas in the same order as the cashier “believes” they paid for it. We could also check whether customers believe they get gas in the same order as they believe they paid for it. (Similarly, we could also check whether the pump believes customers get gas in the same order as the customers believe they paid for it, etc.)

To check this property for the second version, we modified the INCA query and FLAVERS QRE described in Section 4.1 to use the rendezvous in the customer task. INCA found a solution to the inequalities and produced the corresponding behavior of each task. These behaviors yield an execution of the system in which the first customer completes the rendezvous with the connector between it and the cashier, followed by the corresponding rendezvous between the second customer and its connector, but the second customer's connector delivers the money to the cashier before the first customer's connector. (A similar race occurs with the connector between the pump and the customers even if the money arrives at the cashier in the correct order.) FLAVERS produced the same execution.

The problem here is that, while communication between a component and a connector is synchronous, the communication between two components mediated by that connector is not. We can think of it as the customer “mailing” the money to the cashier, and the pump similarly “mailing” the gas to the customer—the customer passes the money into the connector, but has no way of knowing when the connector delivers it to the cashier. This is in contrast to the original Ada versions of the gas station presented by Helmhold and Luckham, where the communication between customers and the cashier was via direct Ada rendezvous between the two tasks.

In a certain sense, of course, this is not a critical requirement for the gas station, since customers do get the gas they pay for. In a real gas station, though, it would certainly make customers unhappy. We therefore decided to modify the architecture to ensure that customers receive gas in the order they pay, as viewed by the customers themselves. There are a number of ways in which such a modification might be carried out. One would be to use a single connector tying both customers to the cashier, and a single connector from the pump to the two customers. Another would be to add additional connectors from the cashier to the customers and from the customers to the pumps, allowing the components to signal when they had received money or gas. Instead, we chose to keep the basic “boxes and arrows” structure, but to modify the components and connectors so that the connectors signal the component that sends information when that information has been delivered. We did this by adding “callback” and “go-ahead” actions to the
communication between the customers and cashier, and between the pump and the customers. The new versions of the customer and cashier tasks and the customer-cashier connectors are shown in Figure 6; the other modifications are similar.

We then analyzed this modified architecture, translating it into Ada in the same way as the first two versions (i.e., with one task for each component and connector, etc.). Now, however, we identified the event of a customer paying for gas with the rendezvous representing the callback from the connector signaling that the money had been delivered to the cashier. As for the previous case, we identified the event of the customer getting gas with the rendezvous between the customer and the customer-pump connector at the customer’s oil pump entry.

For INCA, it was necessary for technical reasons (again involving cycles in the FSAs) to decompose the property into two queries. We first wrote a query to check whether the cashier tells the pump to give gas to the customers in the same order as the customers pay for gas (in terms of the callback rendezvous). INCA verified this property. We then used a query that checked whether customers get gas in the same order as the cashier tells the pump to give it to them. INCA also verified this. Together, these show that customers get gas in the same order as they pay.

Using QREs for the same two subproperties, FLATVERS also verified the property.

We also verified the other properties for this version of the architecture, using both INCA and FLATVERS.

4.5 Performance

INCA and FLATVERS are research prototypes, and so the absolute time that analyses of the properties took are indicative of neither the real potential of the tools nor their scalability. However, we present these times here to illustrate the current state of the tools. We ran all experiments on a DEC Alpha Station 200 with 128 megabytes of physical memory. For each of the three versions of the architecture, it took less than 20 seconds for each of the tools to create the appropriate internal representation used by the analyses. Table 1 gives the time it took each of the tools to check each of the properties discussed in the previous section. (All times are in seconds, and include both user and system time.)

In addition to the application-specific properties, the tools are also capable of checking general properties. For example, we used INCA to prove the absence of deadlock in all three versions of the architecture. (The
Table 1: Time to check the properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Version 1</th>
<th></th>
<th>Version 2</th>
<th></th>
<th>Version 3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INCA</td>
<td>FLAVERS</td>
<td>INCA</td>
<td>FLAVERS</td>
<td>INCA</td>
<td>FLAVERS</td>
</tr>
<tr>
<td>1st Race Cond.</td>
<td>0.8</td>
<td>33.05</td>
<td>0.24</td>
<td>33.29</td>
<td>0.67</td>
<td>163.80</td>
</tr>
<tr>
<td>No Free Gas</td>
<td>0.85</td>
<td>82.09</td>
<td>1.36</td>
<td>347.16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Right Amount</td>
<td>0.77</td>
<td>83.21</td>
<td>0.86</td>
<td>404.32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2nd Race Cond.</td>
<td>0.68</td>
<td>22.15</td>
<td>1.59</td>
<td>95.79</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deadlock</td>
<td>0.44</td>
<td>0.79</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The current implementation of FLAVERS cannot check for deadlock.

5 Conclusions

In this paper, we have shown how existing static analysis tools can be used to check application-specific properties of architectural specifications. The tools were able to detect faults in the specifications, to provide example executions displaying the faults, and to verify that modifications to the specifications correctly removed the faults. Such tools can provide critical early feedback to system architects, helping to reduce the cost and improve the reliability of distributed systems.

While our initial exploration used WRIGHT as the architectural description language and INCA and FLAVERS as the static analysis tools, we see nothing in the approach that limits the approach to a particular language or tools. Although the close relation between CSP and Ada made it easy to manually translate the WRIGHT specification into Ada for use with our tools, we expect that the internal representations that static concurrency analysis tools use could be created from most architectural description language with sufficiently well-defined semantics. Similarly, other static analysis tools capable of formulating and checking application-specific properties, such as SPIN [11] or SMV [17] could be used with architectural specifications.

The static analysis tools automate the checking of properties, but it is still up to the system architect to formulate those properties. As always, this is not straightforward and has to be done carefully. The fact that the tools can provide “counterexamples” when they cannot verify a property can, however, provide important assistance to the architect in understanding complex features of the system.

The preliminary investigation reported here suggests a number of interesting directions for future work. First, analyzing software architectures specified in additional architectural description languages may indicate particular language constructs that affect different kinds of static analysis and may suggest extensions to the existing analysis tools or modifications to the architectural languages in order to achieve improved analysis support. For example, the dynamic features of Darwin might cause difficulties for many static analysis techniques. Another research direction involves the analysis of architectural styles, families of architectures with common structure. Analysis results for an architectural style should be applicable to instantiations of that style. These results could be used to show that an instantiation correctly conforms to a style or perhaps as constraints to improve the accuracy of analysis of an instantiation of that style. Finally, we note that the static analysis tools can be used to show that a refinement or implementation of an architecture has the properties assumed in the architectural description. For instance, the tools could show that the implementation of a connector in a pipe-and-filter architecture actually behaves as a pipe.

The gas station is a small, but relatively rich, example. The race condition in which one customer takes the pump before another customer has been studied from various standpoints in the static concurrency analysis literature, and the two WRIGHT specifications supplied to us by Garlan were intended to illustrate it. The second race condition, arising from the asynchronous communication between components provided by the connectors in the first two versions of the architecture, does not arise in the Ada implementations of the gas station used in earlier concurrency analysis. The static analysis identified a genuine architectural
issue that we, at least, had not expected to encounter. We make no claim, of course, that our third version of the gas station specification is the optimal way to avoid this race, but we believe that the way that the tools detected this unexpected problem and verified that a modification did indeed correct it illustrates the importance of applying static concurrency analysis techniques to architectural descriptions. While analyzing larger and more complex architectures will of course be somewhat harder, the much greater difficulty in understanding those larger and more complex systems makes static analysis even more important.

Acknowledgments  This work was supported in part by the Air Force Materiel Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract F30602-94-C-0137 and in part by the National Science Foundation grant CCR-9407182.

The authors gratefully acknowledge the help of David Garlan in providing WRIGHT specifications for the gas station example.

References


A  Ada Code

A.1  Version One

package Gas is

type AMOUNT is range 1 .. 2;

task Customer1 is
entry Oil_pump ( gas_amount : in AMOUNT);
end Customer1;

task Customer2 is
entry Oil_pump ( gas_amount : in AMOUNT);
end Customer2;

task Cashier is
entry Customer1_pay ( cash_amount : in AMOUNT);
entry Customer2_pay ( cash_amount : in AMOUNT);
end Cashier;

task Pump is
entry From_cashier_pump ( cash_amount : in AMOUNT);
entry Oil1_take;
entry Oil2_take;
end Pump;

task Customer1_cashier is
entry Get_money_pay ( cash_amount : in AMOUNT);
end Customer1_cashier;

task Customer2_cashier is
entry Get_money_pay ( cash_amount : in AMOUNT);
end Customer2_cashier;

task Customer1_pump is
entry Getoil_take;
entry Getoil_pump ( cash_amount : in AMOUNT);
end Customer1_pump;

task Customer2_pump is
entry Getoil_take;
entry Getoil_pump ( cash_amount : in AMOUNT);
end Customer2_pump;

task Cashier_pump is
entry Tell_pump [ cash_amount : in AMOUNT];
end cashier_pump;

task body Customer2 is
begin
loop
Customer2_cashier.Getmoney_pay ( cash );
Customer2_pump.Getoil_take;
accept Oil_pump ( gas_amount : in AMOUNT);
end loop;
end Customer2;

task body Cashier is
begin
loop
select
accept Customer1_pay ( cash_amount : in AMOUNT ) do
    cash := cash_amount;
end Customer1_pay;
cashier_pump.Tell_pump(cash);
end select;
end loop;
end cashier;

task body Pump is
begin
loop
select
accept From_cashier_pump ( cash_amount : in AMOUNT ) do
    cash := cash_amount;
end From_cashier_pump;
end loop;
end Pump;

task body Customer1_cashier is
begin
loop
accept Get_money_pay ( cash_amount : in AMOUNT ) do
    cash := cash_amount;
accept Oil_pump ( gas_amount : in AMOUNT );
end loop;
end Customer1_cashier;

task body Customer2_cashier is
begin
loop
accept Get_money_pay ( cash_amount : in AMOUNT );
accept Oil_pump ( gas_amount : in AMOUNT );
end loop;
end Customer2_cashier;
task body Cashier is
begin
loop
accept Getmoney { cash_amount : in AMOUNT } do
  cash := cash_amount;
  end Getmoney;
cashier.Customer1_pay(cash);
end loop;
end Customer2_cashier;
task body Cashier2_Pump is
cash : AMOUNT;
begin
loop
  accept Tell_pump { cash_amount : in AMOUNT } do
    cash := cash_amount;
    end Tell_pump;
pump.From_cashier_pump(cash);
end loop;
end cashier_pump;
task body Customer1_Pump is
cash : AMOUNT;
begin
loop
  accept Getoil_take;
pump.Oil_take;
  accept Getoil_pump { cash_amount : in AMOUNT } do
    cash := cash_amount;
    end Getoil_pump;
    Customer1.Oil_pump(cash);
  end loop;
end Customer1_pump;
task body Customer2_Pump is
cash : AMOUNT;
begin
loop
  accept Getoil_take;
pump.Oil_take;
  accept Getoil_pump { cash_amount : in AMOUNT } do
    cash := cash_amount;
    end Getoil_pump;
    Customer2.Oil_pump(cash);
  end loop;
end Customer2_pump;
end Gas;

package body Gas is

task body Customer1 is
entry Oil_pump { oil_amount : in AMOUNT };
end Customer1;
task body Customer2 is
entry Oil_pump { oil_amount : in AMOUNT };
end Customer2;
task body cashier is
entry Customer1_pay { cash_amount : in AMOUNT };
entry Customer2_pay { cash_amount : in AMOUNT };
end cashier;
task pump is
entry From_cashier_pump1 { cash_amount : in AMOUNT };
entry From_cashier_pump2 { cash_amount : in AMOUNT };
end pump;
task Customer1_cashier is
entry Getmoney { cash_amount : in AMOUNT };
end Customer1_cashier;
task Customer2_cashier is
entry Getmoney { cash_amount : in AMOUNT };
end Customer2_cashier;
task Customer1_pump is
entry Getoil_pump { cash_amount : in AMOUNT };
end Customer1_pump;
task Customer2_pump is
entry Getoil_pump { cash_amount : in AMOUNT };
end Customer2_pump;
task cashier_pump is
entry Tell_pump1 { cash_amount : in AMOUNT };
entry Tell_pump2 { cash_amount : in AMOUNT };
end cashier_pump;
end Gas;

package body Gas is

task body Customer1 is
entry Oil_pump { oil_amount : in AMOUNT };
begin
loop
  Customer1_cashier.Getmoney { cash };
  accept Oil_pump { oil_amount : in AMOUNT };
end loop;
end Customer1;
task body Customer2 is
entry Oil_pump { oil_amount : in AMOUNT };
begin
loop
  Customer2_cashier.Getmoney { cash };
  accept Oil_pump { oil_amount : in AMOUNT };
end loop;
end Customer2;
task body cashier is
entry Customer1_pay { cash_amount : in AMOUNT };
entry Customer2_pay { cash_amount : in AMOUNT };
end cashier;
task body Pump is
    cash : AMOUNT;
    begin
    loop
      select
        accept Fromcashier.pump1 (cash_amount : in AMOUNT) do
          cash := cash_amount;
          FromCashier.pump1;
          Customer1.pump.Getoil.pump(cash);
        or
        accept Fromcashier.pump2 (cash_amount : in AMOUNT) do
          cash := cash_amount;
          FromCashier.pump2;
          Customer2.pump.Getoil.pump(cash);
      end select;
    end loop;
    end pump;
end Customer1.Cashier is
  cash : AMOUNT;
  begin
  loop
    accept Getmoney.pay (cash_amount : in AMOUNT) do
      cash := cash_amount;
      Getmoney.pay;
      cashier.Customer1.pay(cash);
    end loop;
  end Customer1.Cashier;
end Customer2.Cashier is
  cash : AMOUNT;
  begin
  loop
    accept Getmoney.pay (cash_amount : in AMOUNT) do
      cash := cash_amount;
      Getmoney.pay;
      cashier.Customer2.pay(cash);
    end loop;
  end Customer2.Cashier;
end Cashier.Pump is
  begin
  loop
    select
      accept Tell.pump1 (cash_amount : in AMOUNT) do
        cash := cash_amount;
        Tell.pump1;
        pump.Fromcashier.pump1(cash);
      or
      accept Tell.pump2 (cash_amount : in AMOUNT) do
        cash := cash_amount;
        Tell.pump2;
        pump.Fromcashier.pump2(cash);
      end select;
    end loop;
  end cashier.pump;
end Customer1.Pump is
  cash : AMOUNT;
  end loop;
end Cashier;

package Gas is

  type AMOUNT is range 1 .. 2;

  task Customer1 is
    entry Getmoney.callback;
    entry Oil.pump (oil_amount : in AMOUNT);
    entry Oil.goahead;
  end Customer1;
end Customer2 is
  entry Getmoney.callback;
  entry Oil.pump (oil_amount : in AMOUNT);
  entry Oil.goahead;
end Customer2;
end Cashier is
  entry Customer1.pay (cash_amount : in AMOUNT);
  entry Customer1.goahead;
  entry Customer2.pay (cash_amount : in AMOUNT);
  entry Customer2.goahead;
end cashier;
end pump is
  entry Fromcashier.pump1 (cash_amount : in AMOUNT);
  entry Fromcashier.pump2 (cash_amount : in AMOUNT);
  entry Getoil.callback;
end pump;
end Customer1.cashier is
  entry Getmoney.pay (cash_amount : in AMOUNT);
  end Customer1.cashier;
end Customer2.cashier is
  entry Getmoney.pay (cash_amount : in AMOUNT);
  end Customer2.cashier;
end Customer1.pump is
  entry Getoil.pump (cash_amount : in AMOUNT);
end Customer1.pump;

A.3 Version Three
task Customer2.pump is
  entry Getoil.pump (cash_amount : in AMOUNT);
end Customer2.pump;

task cashier.pump is
  entry Tell.pump1 (cash_amount : in AMOUNT);
  entry Tell.pump2 (cash_amount : in AMOUNT);
end cashier.pump;

end Gas;

package body Gas is

task body Customer1 is
  cash : AMOUNT;
begin
  loop
    Customer1.pump.Getoil (cash_amount);
    accept Getoil_callback;
    accept Oil.pump (oil_amount : in AMOUNT);
    accept Oil.goalhead;
  end loop;
end Customer1;

task body Customer2 is
  cash : AMOUNT;
begin
  loop
    Customer2.pump.Getoil (cash_amount);
    accept Getoil_callback;
    accept Oil.pump (oil_amount : in AMOUNT);
    accept Oil.goalhead;
  end loop;
end Customer2;

end Customer1,
cashier.pump is

end Customer2;

end cashier;

task body Cashier is
  cash : AMOUNT;
begin
  loop
    select
      accept Customer1.pay {cash_amount : in AMOUNT} do
        cash := cash_amount;
        end Customer1.pay;
        or
        accept Customer2.pay {cash_amount : in AMOUNT} do
        cash := cash_amount;
        end Customer2.pay;
        or
        accept Teller.2.pay {cash_amount : in AMOUNT} do
        cash := cash_amount;
        end Teller.2.pay;
    end select;
  end loop;
end Cashier;

end Customer1,
Pump is

end Customer2;

end Pump;

task body Customer1.Pump is
  cash : AMOUNT;
begin
  loop
    select
      accept Fromcashier.pump1 {cash_amount : in AMOUNT} do
        cash := cash_amount;
        end Fromcashier.pump1;
        Customer1.pump.Getoil.pump(cash_amount);
        accept Getoil_callback;
        or
        accept Fromcashier.pump2 {cash_amount : in AMOUNT} do
        cash := cash_amount;
        end Fromcashier.pump2;
        Customer2.pump.Getoil.pump(cash_amount);
        accept Getoil_callback;
    end select;
  end loop;
end Pump;

end FromCashier.pump1;

end FromCashier.pump2;

end Customer1;
end Customer2;

end task body Customer1.Cashier is
  cash : AMOUNT;
begin
  loop
    accept Getmoney.pay {cash_amount : in AMOUNT} do
      cash := cash_amount;
      end Getmoney.pay;
      cashier.Customer1.pay(cash);
      Customer1.Getmoney_callback;
      cashier.Customer1.goalhead;
    end loop;
end Customer1;

end Customer2;

end task body Customer2.Cashier is
  cash : AMOUNT;
begin
  loop
    accept Getmoney.pay {cash_amount : in AMOUNT} do
      cash := cash_amount;
      end Getmoney.pay;
      cashier.Customer2.pay(cash);
      Customer2.Getmoney_callback;
      cashier.Customer2.goalhead;
    end loop;
end Customer2;

end Cashier;

end task body Cashier.Pump is
  cash : AMOUNT;
begin
  loop
    select
      accept Teller.1.pay {cash_amount : in AMOUNT} do
        cash := cash_amount;
        end Teller.1.pay;
        or
        accept Teller.2.pay {cash_amount : in AMOUNT} do
        cash := cash_amount;
        end Teller.2.pay;
    end select;
  end loop;
end Cashier;

end Teller.1;
end Teller.2;

17
Customer1.Oil|pump(cash);  
pump.Getoil|callback;  
Customer1.Oil|gohead;  
end loop; 
end Customer1|pump;

task body Customer2|pump is 
cash : AMOUNT;  
beg
loop
accept Getoil|pump (cash|amount : in AMOUNT) do 
cash := cash|amount;  
end Getoil|pump;  
Customer2.Oil|pump(cash);  
pump.Getoil|callback;  
Customer2.Oil|gohead;  
end loop;  
end Customer2|pump;  
end Gas;

B Property Specifications

B.1 The Critical Race to the Pump

For the first version:

INCA

(defquery "race" "nofair"
  (omega-starless
   (sequence
    (interval :initial t :open t
     :ends-with '(
      (rend "customer1_cashier;cashier.customer1_pay")))
    (interval :ends-with '(
      (rend "pump;customer2_pump.getoil"))
     :require '(
      (rend "customer2_cashier;cashier.customer2_pay"))
     :forbid '(
      (rend "pump;customer1_pump.getoil"))))))

FLAVERS

{cashier_customer1_pay, cashier_customer2_pay, 
customer1_pump_getoil_pump, customer2_pump_getoil_pump}

none

.:; 
cashier_customer1_pay;  
[-customer1_pump_getoil_pump] *; 
cashier_customer2_pay;  
[-customer1_pump_getoil_pump] *; 
customer2_pump_getoil_pump;  
.*

For the second version:

INCA

(defquery "first-race-v2-1" "nofair"
  (omega-starless
   (sequence
    (interval :initial t :open t
     :ends-with '(
      (rend "customer1_cashier;cashier.customer1_pay")))
    (interval :ends-with '(
      (rend "cashier;cashier_pump.tell_pump2"))
     :forbid '(
      (rend "cashier;cashier_pump.tell_pump1"))
     :require '(
      (rend "customer2_cashier;cashier.customer2_pay"))))))

(defquery "first-race-v2-2" "nofair"
  (omega-starless
   (sequence
    ....

18
(interval :initial t :open t
  :endswith '(read "cashier; cashier_pump.tell_pump1")))
(interval :endswith '(read "pump; customer2_pump.getoil_pump")
  :forbid '(read "pump; customer1_pump.getoil_pump")
  :require '(read "cashier; cashier_pump.tell_pump2"))))

FLAVORS
Same as for the first version

B.2 No Free Gas

INCA
(defquery "free-gas-1" "nofair"
  (omega-star-less
   (sequence
    (interval :initial t :open t
     :constraints '(
      (<= (+ 1 "call(customer1; customer1_cashier.getmoney_pay;1)"
      "accept(customer1_pump; customer1.oil_pump;1)"))))
    (interval :ends-with 'read "pump;;customer1_pump.getoil_pump;;2")
    :forbid '(read "pump;;customer1_pump.getoil_pump;;1")
    :require '(read "cashier; cashier_pump.tell_pump2")))))

FLAVORS
{cashier_customer1_pay, customer1_pump.getoil_pump}

all
(cashier_customer1_pay; customer1_pump.getoil_pump)*

B.3 Customers Get the Right Amount of Gas

INCA
(defquery "correct-amount" "nofair"
  (omega-star-less
   (sequence
    (interval :initial t :open t
     :endswith '(read "cashier_customer1; cashier_customer1_pay;1")
    (interval :endswith '(read "pump; customer1_pump.getoil_pump;2")
     :forbid '(read "pump; customer1_pump.getoil_pump;1")
     :require '(read "cashier; cashier_pump.tell_pump2")))))

FLAVORS
{cashier_customer1_pay-1, cashier_customer1_pay-2, cashier_customer2_pay-1, cashier_customer2_pay-2, customer1_pump.getoil_pump-1, customer1_pump.getoil_pump-2, customer2_pump.getoil_pump-1, customer2_pump.getoil_pump-2}

none

.*;
B.4 Another Race Condition

For the second version:

INCA

(defquery "second-race-v2" "nofair"
 (omega-star-less
  (sequence
   (interval :initial t :open t
     :ends-with '(
       (rend "customer1_cashier:customer1_getmoney_pay")))
   (interval :ends-with '(rend "customer2_pump:customer2_oil_pump"))
     :require '(rend "customer2_cashier:customer2_getmoney_pay")
     :forbid '(rend "customer1_pump:customer1_oil_pump"))))

FLAVERS
{customer1_cashier_getmoney_pay, customer2_cashier_getmoney_pay, customer1_oil_pump, customer2_oil_pump}

none

.*;

customer1_cashier_getmoney_pay;
[-customer1_oil_pump]*;

For the third version:

INCA

(defquery "second-race-v1" "nofair"
 (omega-star-less
  (sequence
   (interval :initial t :open t
     :ends-with '(rend "customer1_cashier:customer1_getmoney_callback")))
   (interval :ends-with '((rend "cashier:cashier_pump.tell_pump2"))
     :forbid '(rend "cashier:cashier_pump.tell_pump1")
     :require '((rend "customer2_cashier:customer2_getmoney_callback"))))

(defquery "second-race-v2" "nofair"
 (omega-star-less
  (sequence
   (interval :initial t :open t
     :ends-with '(rend "cashier:cashier_pump.tell_pump2"))
     (rend "customer1_cashier:customer1_getmoney_callback")))
   (interval :ends-with '(rend "customer2_pump:customer2_oil_pump")
     :require '((rend "cashier:cashier_pump.tell_pump2"))
     :forbid '(rend "customer1_pump:customer1_oil_pump")
     (rend "cashier:cashier_pump.tell_pump2"))))

FLAVERS
{customer1_getmoney_callback, customer2_getmoney_callback, }
cashier_pump_tell_pump1, cashier_pump_tell_pump2

none

.*:
customer1_getmoney_callback;
[-cashier_pump_tell_pump1]*;
customer2_getmoney_callback;
[-cashier_pump_tell_pump2]*;
customer2_oil_pump;

.

{ cashier_pump_tell_pump1, cashier_pump_tell_pump2, customer1_oil_pump, customer2_oil_pump}

none

.*:
cashier_pump_tell_pump1;
[-customer1_oil_pump]*;
cashier_pump_tell_pump2;
[-customer1_oil_pump]*;
customer2_oil_pump;

.

B.5 Deadlock

INCA

(defquery "deadlock" "nofair"
  (omega-star-less
   (sequence
    (interval :initial t :progress t :costs "connect-arc-unit"))))