Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Tomáš Poch

Distributed Behavior Protocol Checker

Department of Software Engineering
Supervisor: RNDr. Jan Kofroň
Study Program: Computer Science, Software Systems

I would like to thank to my advisor for his valuable comments and suggestions. His experience with checking behavior protocols helped me a lot. I also want to thank to Ondřej Šerý for important feedback and discussions. Finally I want to thank to my family for their support.

In Prague on October 20, 2006 Tomáš Poch

Contents

Název práce: Distributed Behavior Protocol Checker
Autor: Tomáš Poch
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Jan Kofroň
e-mail vedoucího: `kofron@nenya.ms.mff.cuni.cz`

Abstrakt:

Nárůst dostupné výpočetní síly umožnil v posledních letech praktické využití formální verifikace softwarových systémů. Nejpalčivějším problémem, který zabraňuje širšímu využití však zůstává velikost stavových prostorů. Proto jsou tyto techniky zatím omezeny na relativně malé úlohy. Jednou z možností jak podstatně snížit počet stavů je modelování softwaru pomocí behavior protokolů. [1] Jedná se o regulární výrazy, které popisují chování softwarových komponent. Konkrétní implementace komponenty je tedy ověřena jen jednou oproti protokolu a při verifikaci celé aplikace je již skryta. Ta se pak redukuje na ověření toho, že protokoly komponent ze kterých se aplikace skládá k sobě pasují. Nicméně velikost i takto zjednodušeného stavového prostoru bývá typicky exponenciální vzhledem k délce popisu modelu.

Distribuovaný průchod stavovám prostorem společně s jeho generováním 'za letu' [2] by měl ještě více rozšířit rozsah problémů zvládnutelných touto technikou.

Klíčová slova: model checking, behavior protocol, komponenty

Title: Distributed Behavior Protocol Checker
Author: Tomáš Poch
Department: Department of Software Engineering
Supervisor: RNDr. Jan Kofroň
Supervisor's e-mail address: `kofron@nenya.ms.mff.cuni.cz`

Abstract:

Growth of the computability power in the last years enabled practical use of model checking of software systems. However the state space explosion is still a burning problem that limits usage of this technique to the relatively small tasks. One of the approaches that significantly decrease state space of the task is Behavior Protocol [1]. Behavior protocol is regular language that describes behavior of software component so that component implementation details are hidden during checking of whole application - what is reduced to the checking whether behavior protocols of used components are compliant. However even checking of behavior protocols compliance faces the exponential growth of number of states.

Distributed state space traversing together with 'on the fly' state space generation [2] can be used to improve both time and space requirements.

Keywords: model checking, behavior protocol, components

Chapter 1

Introduction

As the software systems became an essential part of many kinds of human activities in the last few decades the requirements on these systems has changed. The amount of developed software is increasing, the solved tasks are more complex and user requirements changes more often. Supported by permanent decreasing of computational power costs the hardware requirements are not on the first place any more. The software producers are more concerned about development costs as the people who develop the software are more valuable than hardware used to run their applications.

One of the ways to decrease the work needed to develop and maintain an application is a component paradigm. The application is built from well-defined pieces of software called components. This increases maintainability of software and enables re-usability of components in other applications.

The next factor that significantly increases the cost of software development are errors. The errors are expensive especially when they are found in later stages of the development process. *Model-checking* techniques allow developers to check whether their product satisfies particular properties. The errors that are covered by the properties can be found relatively soon. This approach is stronger than testing. It provides the proof that the tested system is correct.

The *model-checking* is often based on traversing of state space. As the state spaces of software systems are typically exponential, the *model-checking* becomes extremely resource demanding task. It significantly decreases range of applications that can be checked with reasonable resources.

1.1 Components

Software components are the next step in the methodology of programming. Coming after object oriented programing, this paradigm provides stronger encapsulation.

The components are meant as building blocks of applications. Once a component is written it can be reused in many applications. Moreover, components can be nested which means that more complex components can be built by connecting simpler ones. A component defines not only the methods it provides to other components, but it also holds information about services required to accomplish its tasks. Group of methods that are associated with some functionality is called *interface.* Each *interface* is *provided* or *required* by the component.

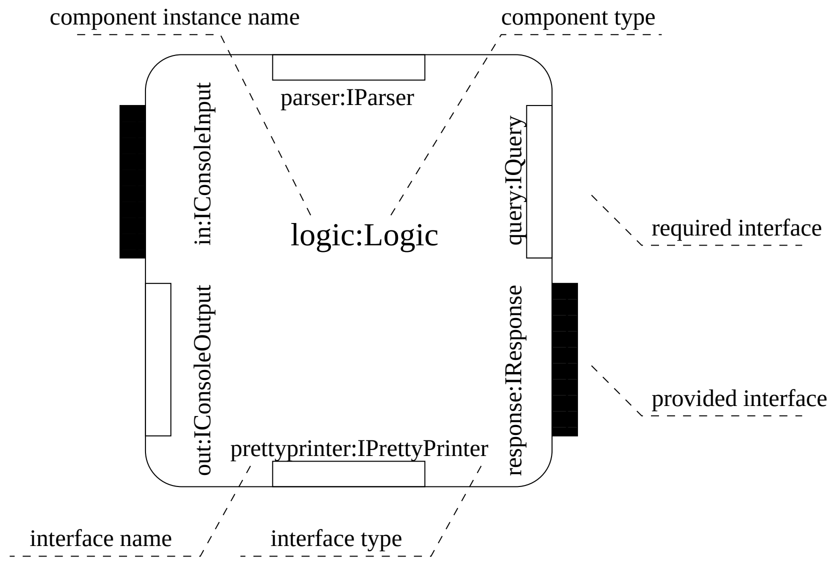


Figure 1.1: Component in detail

The Fig. 1.1 depicts an example of component that *provides* `in` interface of type `IConsoleInput`. This interface contains `inputLine` method that can be invoked by `IConsoleInput` implementation. `IConsoleInput` implementation bound to this interface uses it to pass a line entered by user on console to the component for processing. Once the component obtains a line it uses a method of *required* `parser` interface to ask a component that implements `IParserInterface` to parse the line. Then the returned parse tree is passed to a component bound to `query` interface and so on.

In the component implementation the other components are referenced only by interface names so the binding can be easily changed without recompilation or even changes in the code.

An application is built by binding provide interfaces to require interfaces. A *composite* component is constructed almost the same way. Its provide interfaces are delegated to interfaces of sub-components. Sub-components' required interfaces that are not provided by other subcomponents on the same level are subsumed to the *composite* component's required interfaces. This terminology comes from SOFA [3] component framework. Other frameworks may use other terms to refer to the same principles.

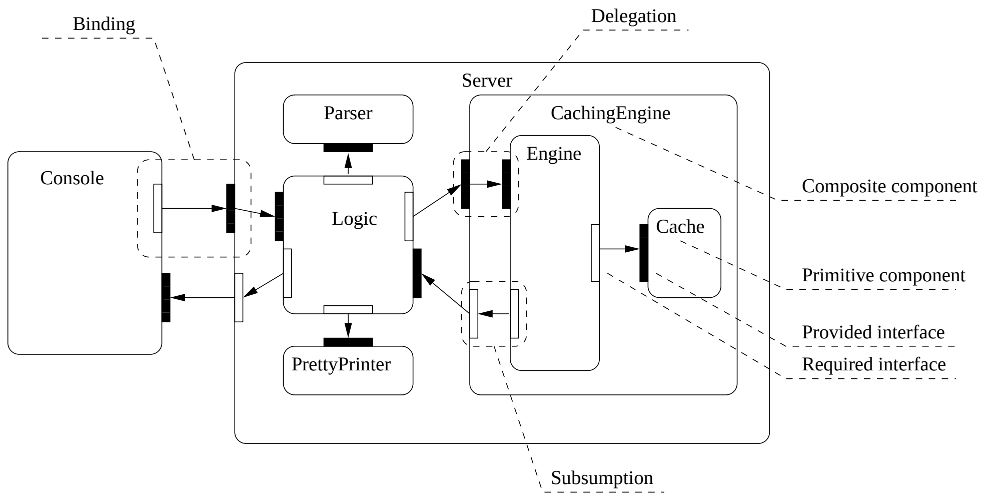


Figure 1.2: Example of component application. The `Console` component is *primitive*, while the `Server` component is *composite*. The `Server` component contains four other components to achieve required functionality — answering questions from the console.

The component frameworks often allow the user to deploy the components an application consists from to different machines to improve performance and reliability. However, the components' source code looks still the same regardless of the application is going to be distributed or not.

The typical usage of component methodology in software development relies on the set of ready-to-use components that the developer team may use. These components can be created by the team during the work on previous projects or even bought from another vendor. As well as a construction engineer does not design building panels again and again for different buildings and an electrical engineer uses transistors and chips produced by companies from different continents, software developers should built new applications from components they already have with only a few new components developed for special needs of a particular project.

1.2 Model checking

Model checking is a technique that takes a model of a system and a set of properties and says whether the model meets the properties. The model is typically derived from software or hardware design.

There is a wide range of formalisms used to express the models. Some model checkers [4] accept models in languages originally developed for the model checking, such as Promela. As the model checking was initially used in hardware development, there are model checkers designed to check models in hardware description languages such as VHDL or Verilog [5]. To get these methods closer to practical software developers, there are even quite successful attempts to use general programming languages such as Java [6] for modeling. This allows direct model checking of the code. However, there are still some limitations.

The model checkers may have a predefined set of generic properties, such as absence of deadlocks, they are able to check. In better cases, they allow the user to specify its own properties. These properties are expressed by temporal logic or a kind of assertions in the model description.

There are two main groups of model-checkers. Explicit model checkers exhaustively enumerates all states of the model. It is checked whether each state complies to all properties to be checked. The second group — symbolic model checking tries to work with many similar states at once. *Binary Decision Diagram — BDD* is a structure often used in symbolic model-checking.

The model checking process is based on the traversal of state space described by the model. These state spaces are often huge because the state space size is typically exponential to the size of a model description. The state space explosion is the most serious problem of the model checking that does not allow it to become widely used by software developers. The state space explosion allows us to check relatively small tasks. The bigger projects are not able to be checked with reasonable resources.

The situation in hardware design is much better. The systems do not have so many states so that model checking already has a practical usage in this area [7].

1.3 Behavior protocols

The main contribution of *behavior protocols* is usage of component paradigm for splitting the model into smaller parts. These parts are checked independently. As the state space is exponential to the model size, the splitting of a model brings a significant reduction of state space size.

The basic idea is to describe behavior of each component by regular expression that captures the traffic on the component boundaries. These expressions are called behavior protocols.

Having the component behavior described by behavior protocols, three types of correctness checking can be performed: implementation-to-protocol obeying, compliance test, and composition test.

However, even the checking of protocol compliance and composition is exponential and thus resources demanding task.

1.4 Goals

The goal of this thesis is to implement a tool for performing compliance and composition tests. The tool will be able to be executed in a distributed way and exploit thus the power of more computer nodes.

As the part of the thesis, a new efficient state space representation must be created, because the representation used by previous version of behavior protocol checker does not fit well to the needs of distributed environment.

This work is not concerned in code analysis and checking implementation of primitive components.

1.5 Structure of this work

The second chapter gives brief introduction into behavior protocols. Additionally to the grammar and semantics of behavior protocols, the properties that can be checked are explained.

The next chapter is focused on the principles of *behavior protocol* checking and approaches used in previous checkers. Also ideas behind the distributed version of checker are presented.

The fourth chapter describes the implementation of checker while the fiveth chapter presents the achieved results. The next chapter contains related work and the last chapter offers future work topics and conclusion.

Chapter 2

Behavior Protocols

This chapter gives the reader a brief introduction into *behavior protocols*. *Behavior protocols* are described formally and in more details in [1].

Behavior protocols are a high-level abstraction used to model communication traffic on component interfaces. *Behavior protocol* is a regular expression used to define the set of all possible finite sequences of method invocations performed through all interfaces of given component.

As *behavior protocols* are a high-level abstraction, they do not hold information about method parameters. This fact is twofold. First, it significantly reduces a model state space, which is exactly what we want from a high-level abstraction, and, second, it allows us to use simple formalism to express protocols - regular expressions. On the other hand, method parameters often influence control flow, *behavior protocols*, however, are not capable to express this property.

2.1 Model description

The model of an application consists of one *behavior protocol* for each component involved and information about interface bindings and components' nesting.

2.1.1 Language

A *behavior protocol* consists of event tokens and operators. Events are used to express method invocation. Each event consists of an interface name, a method name and a type. We distinguish following types of events:

- emit of a method invocation request - `!interface.method`$\uparrow$

- accept of a method invocation request - `?interface.method`$\uparrow$
- emit of a method invocation response - `!interface.method`$\downarrow$
- accept of a method invocation response - `?interface.method`$\downarrow$

A method call is represented by four events. First, invoking component Emmit's a request. The request is accepted by the target component. The target component emits a response and, finally, invoking component accepts the response. However, such level of details is not necessary in many cases.

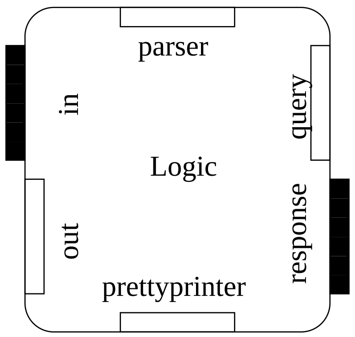
Following abbreviations are available.

- accept method invocation
  `?i.m` is an abbreviation for `?i.m`$\uparrow$`;!i.m`$\downarrow$. A component accepts a method request and immediately emits a response.
  `?i.m{`*`prot`*`}` is an abbreviation for `?i.m`$\uparrow$`;`*`prot`*`;i.m`$\downarrow$ A component accepts a method request, performs some actions described by protocol *prot* and finally emits a response.
- emit method invocation
  `!i.m` is an abbreviation for `!i.m`$\uparrow$`;?i.m`$\downarrow$. A component invokes a method and waits for the response.
  `!i.m{`*`prot`*`}` is an abbreviation for `!i.m`$\uparrow$`;`*`prot`*`;?i.m`$\downarrow$. A component invokes a method and before the response is accepted it can perform some other operations.

The simplest behavior protocol contains only one event.

Following operators are used to construct more complicated protocols. Their priority is defined by grammar in Appendix B.

- alternative operator - `A + B`
  Resulting protocol captures all sequences that are captured by `A` or `B`.
- sequence operator - `A ; B`
  Resulting protocol captures all sequences that consist of a sequence captured by `A` followed by a sequence captured by `B`.
- and-parallel operator - `A | B`
  Resulting protocol captures all sequences that are created by interleaving of an arbitrary sequence from `A` and `B`. This operator is not typical for regular expressions, but it does not make the language stronger. It is an abbreviation only, as the same protocol can be expressed by a combination of alternative and sequence operators.



```
(?in.newLine{!parser.parse;!query.query};
?response.result{prettyprinter.print;!out.putLine})*
```

Figure 2.1: Component with assigned protocol

- or-parallel operator - `A || B`
  This is an abbreviation for `A + B + A|B`

- repetition operator - `A*`
  Resulting protocol captures all finite sequences of sequences from `A` and empty sequence.

- `NULL` operator
  This is nullary operator. Result protocol accepts only the empty sequence.

Having the syntax explained, we can return to our component example enriched by *behavior protocol*.

The *behavior protocol* on Fig.2.1 says that in the initial state the component accepts the invocation of `newLine` method from `in` interface. Then, it calls `parse` method on the `parse` interface followed by calling `query` on the `query` interface. After all of this is done, the component sends the result of calling `newLine` method. Then, the component waits until the `result` method on the `response` interface is invoked. After accepting this invocation, the `print` method on the `prettyprinter` interface is called. Finally, the result is written to the console via the `putLine` method on the *required* `out` interface. As this protocol is wrapped by an instance of the repetition operator, after reporting the result on console the component is ready to accept next query.

Many useful component properties can be observed from this protocol. For example, the component is not able to accept next line from the console until it responds to the last one.

2.2 Finite state machines

As a behavior protocol is a kind of regular expression, it can be represented by *finite state machine* - *FSM* [8]. Although this representation is not very useful for creating particular protocols of real components, it serves well for theoretical and illustration purposes. Even protocol checking is based on it.

Definition 1 *Protocol FSM -* $PFSM_A$ *is a finite state machine accepting exactly the set of sequences generated by the protocol A.*

Examples of *protocol FSM* representing simple protocols are in Appendix A.

2.3 Composition of behavior protocols

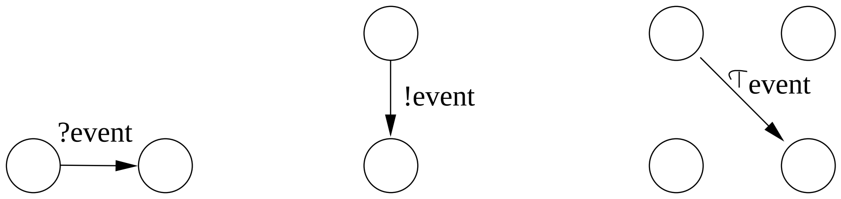
Because our goal is to check the interplay of components, we must be able to obtain a behavior description of multiple connected components. The *consent operator* is a binary operator over *behavior protocols* languages. The *consent operator* is parameterized by list of method names. Result of the operator is a *behavior protocol* capturing behavior of two components *synchronized* on the methods from the parameter list.

We denote a *consent operator* used on the *behavior protocols* `A` and `B` and parameterized with `Sync` which is set of synchronized methods as A $\bigtriangledown^{Sync}$ B.

The *consent operator* was not mentioned in the chapter devoted to the behavior protocol language because a developer can not use it. It is used just by the checker to put the protocols together.

Synchronization means that two *complementary* events of the form `!event` and `?event` are combined into a single event τ `event`. The idea behind this is that emitting and receiving of an event is atomic. As these two actions are atomic and no other action can take place between them we can replace them with so called *tau event* - τ `event`. The *tau event* represents internal action that can not be observed by other component in the system. We denote this *method synchronization* if all events used to express a method invocation are *synchronized*.

As behavior protocols can be represented by *FSM*, we can define the consent operator over finite state machines - See Fig 2.2. We call this FSM *Consent FSM*. In fact, all implementations of consent operator in previous behavior protocol checkers were done using finite state machines. Precise definition of *consent operator* can be found in [9].



(a) (b) (c) Consent of previous protocols

Figure 2.2: *FSM* notion of consent operator.

By combining protocols of all components that are used to form a composite component, we get the *architecture protocol*. The example is shown at Fig.2.3.

2.4 Model properties

There are three general composition errors that may occur during composition of components and can be detected using *behavior protocols* - *bad activity*, *no activity* and *infinite activity*. By checking the application model we ensure that none of these errors is present in the system. Recently user defined application specific properties were introduced in [10]. The *Linear Temporal Logic* - LTL is used to specify them. However distributed checker does not support these properties.

2.4.1 Bad activity

Bad activity occurs when a component tries to call a method and no one accepts it. This can be caused by the fact that there is no component bound to the interface or that the bounded component is not in the state that allows it to accept the call. The first case can not happen if all interfaces are properly bound. [1]

Consider the example on the picture 2.4 with the *behavior protocol* of the `Console` component changed to `(!in.newLine+?out.putLine)*`. This protocol allows `Console` to send another input line to server before the response

[1]We allow incomplete bindings. This can be useful for component reusability. If the user is sure that in given environment a component won't attempt to call a method on its required interface it is useless to bind the interface.

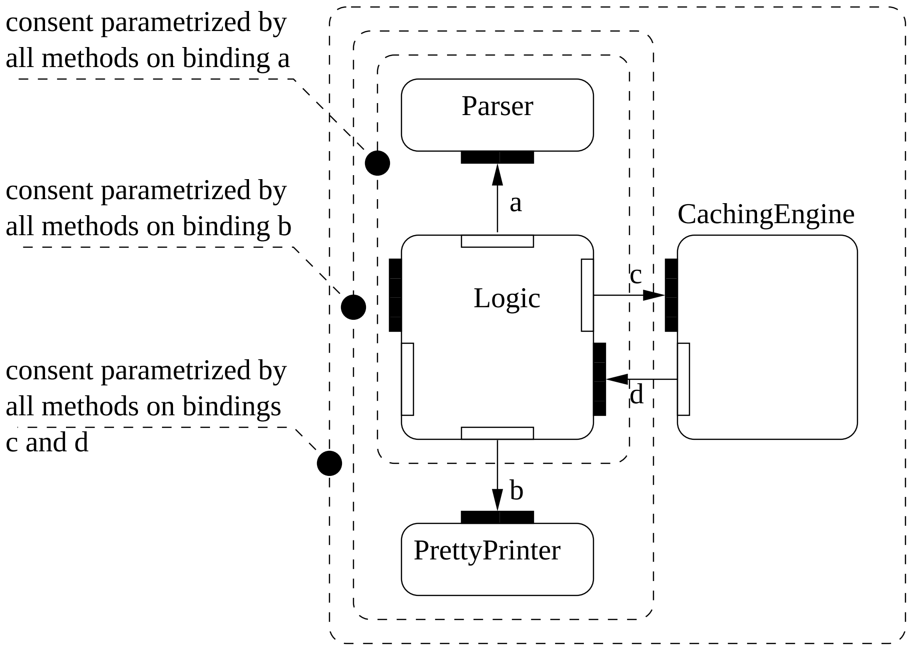
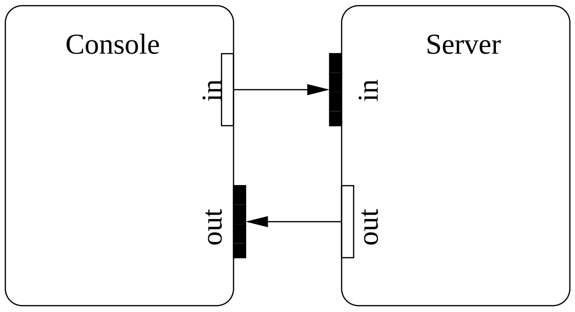

consent parametrized by all methods on binding a
consent parametrized by all methods on binding b
consent parametrized by all methods on bindings c and d
Parser
a
CachingEngine
Logic
c
d
b
PrettyPrinter

Figure 2.3: Iterative usage of *consent* operator in order to obtain *architecture protocol*. At first protocols of components `Parser` and `Logic` are connected by *consent* operator parametrized by all methods from binding Then the result is combined with protocol assigned to `PrettyPrinter`. Finally protocol of `CachingEngine` component is added by *consent* operator parametrized by all methods from bindings `c` and `d`.



Console: `(!in.newLine;?out.putLine)*`
Server: `(?in.newLine;!out.putLine)*`
Consent: $(\tau$`in.newLine;`τ`out.putLine)*`

Figure 2.4: Consent operator applied to two top-level components. To keep the example simple, corresponding interfaces have the same name in both components which is not true in a general case.

to the previous one comes. But the server does not accept such a call before it responds to the previous request. Thus, we get a *bad activity* composition error.

2.4.2 No activity

A component is in a *final state* if a sequence of calls on its boundaries is captured by an associated protocol. A system is in the *final state* if all its parts are in their *final states*

No activity composition error, often referred to as a 'deadlock', occurs when the system is not in a *final state* and at all its parts waits for events but no component emits such an event. Thus, the system can never get to a *final state.*

Let's consider another modification of components on picture 2.4. The `Console` component stays unchanged, but the `Server` component's *behavior protocol* is changed to `(?in.newLine;?in.newLine;!out.putLine)*`. The modified component expects `Console` to sent an request for two new lines before printing out the result. But `Console` doesn't send the second line, because it waits for an invocation of the putLine method. None of the components is in the final state, so the system contains the *no activity* composition error.

2.4.3 Infinite activity

Finally, the infinite activity composition error is known as 'livelock'. In contrast to *no activity*, if an *infinite activity* occurs, the system still performs actions. However, as well as for *no activity*, there is no way how to reach a final state.

The *infinite activity* is defined using FSM representation of behavior protocols. The system contains *infinite activity* composition error, if the FSM representing its behavior contains a cycle reachable from the initial state and there is no path from any state of the cycle to a final state.

This composition error did not prove to be very useful. Similar behavior is sometimes even desired. For this reason and because the checker would become more complicated, checking of this composition error is not implemented by distributed checker.

2.5 Checking the model part by part

Checking of whole application is performed in several steps. The component design of an application yields a hierarchy of components. Its leaves are *primitive components* implemented in a programming language. They are used to form *composite* components and the top of this structure is the entire application. From this point of view, an application is a component tree. Fig. 2.5 shows the tree of an application containing two top-level components. One of them is primitive.

The checking of primitive components is performed at first. In this case, the code must be analyzed to ensure that the protocol captures all possible sequences of method invocations generated by the code.

Having checked all children of a composite component, its subcomponents, we can check the component itself. This involves two steps. First, protocols of all subcomponents are combined using *consent* operator to form the *architecture protocol*. This is called *horizontal compliance* test. Components on the same level are combined and *consent* operator is capable to detect *composition errors* after being applied. Then internal traffic, τ *events*, involved in the combined protocol is omitted and the resulting protocol is checked against the protocol associated with the composite component. This is called *vertical compliance* because protocols from different levels of the component hierarchy are involved. If the tests yield no composition errors, the composite component is successfully checked and we can use it for checking the next level of hierarchy. Notice that during traversing the hierarchy up, the implementation details of underlying components are being forgotten or hidden. On the top level, only *horizontal compliance* test is performed.

compliance test `b`

Logic:	`(?newLine{!parse;!query};`
	`?result{!print;!putLine})*`
CachingEngine:	`(?query;!result)*`
Parser:	`(?parse)*`
PrettyPrinter:	`(?print)*`
Server architecture prot.:	`(?newLine{τparse;τquery};`
	`τresult{τprint;!putLine})*`
Tau events ommited:	`(?newLine;!putLine)*`

composition test `c`

Server:	`(?newLine;!putLine)*`
Console:	`(!newLine;?putLine)*`
Top-level arch. prot.:	`(τnewLine;τputLine)*`

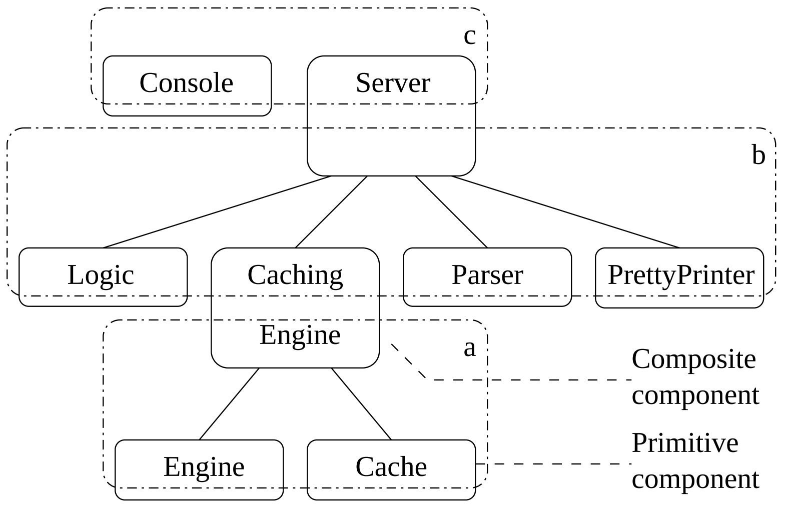


Figure 2.5: Checking of the application from Fig. 1.2. First, the code analysis of *primitive* components is performed. Then, compliance test `a` ensures that components `Cache` and `Engine` can cooperate without a composition error and together complies to the protocol of *composite* component `CachingEngine`. This test, as well as test `b` involves both - *horizontal* and *vertical* compliance. Because all components used by component `Server` are checked at this moment, we can continue with compliance test `b`. The checking process is finished by *horizontal compliance test* `c`.

Chapter 3

Checking the protocols

3.1 Checking of an application

As mentioned in the previous chapter, the checking of a system described by *behavior protocols* consists of three different steps — *primitive component* analysis, *horizontal compliance* test and *vertical compliance* test. This section briefly touches the problems that have to be solved by *primitive component* analysis. Then, as it is the main scope of this work, latter two steps are examined much closer.

3.1.1 Primitive component analysis

The goal of primitive component analysis is to check whether the implementation corresponds to the given protocol. It involves understanding of the language used to implement the component and checking whether each sequence of calls is captured by associated protocol. Notice that programming languages used to implement components are always stronger than regular expressions. The sets of method call sequences generated by the implementation are thus quite complex. To capture all sequences generated by the implementation the associated *behavior protocol* usually includes many sequences that the underlying implementation does not produce in fact.

Let us have a component that accepts an arbitrary number of method `a` calls and then the same number of method `b` calls. To express this behavior exactly, we need expressing power of a stack automaton. As the regular languages does not have such power, we must write a protocol that accepts sequence of method `a` invocations and then sequence of method `b` calls. But these sequences need not have the same length.

Another gap between sequences generated by an implementation and a protocol is caused by absence of method parameters.

Implementing source code analysis would be pretty hard and language dependent. Moreover similar task is already solved by other model checkers. So, the approach chosen to solve this problem is based on exploiting functionality of existing model checkers of conventional programming languages. For checking of components written in Java programming language, Java PathFinder [6] is used. This model checker is quite modular. It allows an user to specify special actions that are performed at each state. Therefore the PathFinder can be combined with *behavior protocol* checker. Combination of behavior protocol state space and state space generated by PathFinder from Java component implementation is examined. This approach is explained in more details in [11].

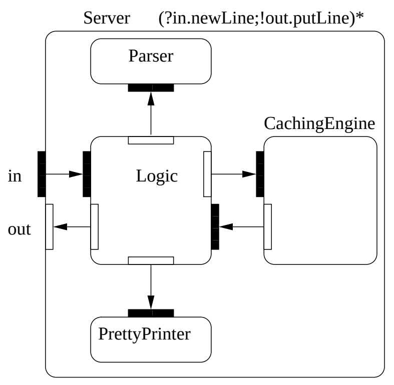
3.1.2 Horizontal compliance

The *horizontal compliance* test is based on the construction of *architecture FSM* that represents behavior of connected components. This *architecture FSM* is another representation of *architecture protocol* already presented.

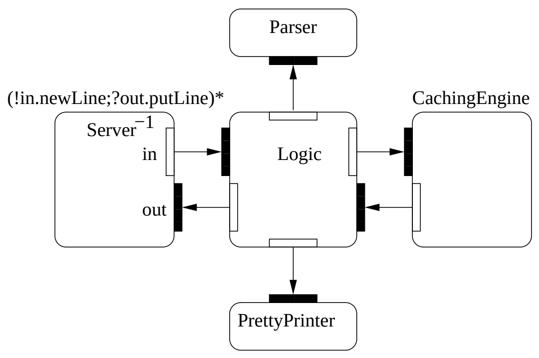
Architecture FSM is created from *FSM*s of behavior protocols the architecture consist from. During the construction of *architecture FSM* all possible combinations of component states are visited and examined. The main problem of this technique is the size of the resultant automaton. While the number of states of *FSM*s created from the protocols the architecture consists from are often small, the *architecture FSM* can be bigger. Its size can grow exponentially.

3.1.3 Vertical compliance

The *vertical compliance* test can be easily transformed into *horizontal compliance* test using the *inverted protocol*[12]. The *inverted protocol* P^{-1} of a protocol P is created from the protocol P by changing all emit events to accept events and vice versa. If we want to test whether the *architecture* protocol corresponds to the protocol of *composite* component, all we must do is inverting the protocol of the *composite* component and then run the *horizontal* compliance test. By inverting the protocol, we create an artificial component on the same level as other components. This component exactly generates the traffic that can occur on the *composite* component boundaries. Note that we can do both steps — *horizontal* and *vertical compliance* test of the *composite* component at once.



(a) Composite component to be checked



(b) Result of the Server protocol inversion

Figure 3.1: Example of protocol inversion. Note the change of required interface to provided interface and vice versa.

3.2 Preprocessing

Preprocessing can be used to hide information about binding into the interface and method identifiers. During the checking itself, these identifiers are treated just like simple strings. Only their equality or inequality is important. Because the set of identifiers does not change during the checking, they can be replaced by integer indexes into a string table. It is useful to hide as many information as possible into equality or inequality of method identifiers. Many special cases, such as multiple binding, can be treated this way.

For the purposes of checking the protocol, we distinguish three classes of architecture descriptions — *gramatically correct architecture description*, *semantically correct architecture description* and *canonical architecture description.*

3.2.1 Grammatically correct architecture description

Definition 2 *Let $P_1, P_2, \ldots, P_N$ are behavior protocols, $Sync_1$, $Sync_2$, ..., $Sync_{N-1}$ and Unbound are sets of methods used in the protocols. Then the sequence "P_1 $Sync_1$ P_2 $Sync_2$ P_3 $Sync_3$ $\ldots P_{N-1}$ $Sync_{N-1}$ P_N Unbound" is* grammatically correct architecture description.

The *grammatically correct architecture description* is a list of *behavior protocols* interleaved with information about synchronized events. Unbound is a set containing all methods from unbound interfaces. Invoking such method means *bad activity*. As it was already mentioned, method calls are modeled using events. There are abbreviations that often hide the details, but there is no grammar rule that forces the user to keep method invocation semantics. So, there are some *grammatically correct protocols* that capture sequences that can not appear on any component system. Such protocols are just wrong. The checker can decide whether they are correct or not, but this information is useless. The protocols do not correspond to the implementation.

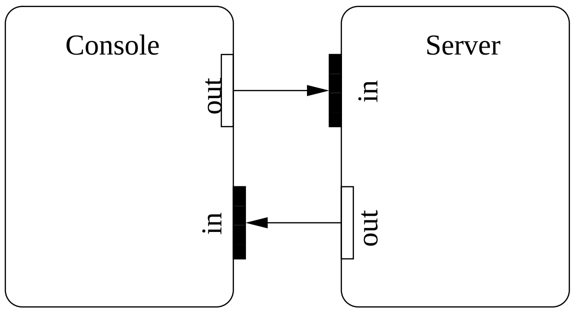
3.2.2 Semantically correct architecture description

A *semantically correct architecture description* is a *grammatically correct architecture description* containing only *semantically correct behavior protocols*. A *semantically correct behavior protocol* is a *behavior protocol* that for each emitted request accepts a response and for each accepted request emits a response. Using only abbreviations, a developer cannot create a *behavior protocol* which is not semantically correct.

These protocols almost capture valid behavior of component systems except for one detail. The invocations expressed in *behavior protocols* do not keep any information about threads. Because there is no rule that prevents it, it is possible that emit event of an method invocation is emitted by one thread and corresponding response event is accepted by another thread. It does not matter, whether these two threads run within one component or not. To avoid such situation, we must disallow both — multiple bindings and invocation of the same method in two threads running simultaneously within one component. Fortunately, there exist a set of rules that enables fully automatic replacement of these constructs. These are the same as the rules used during the transformation of ADL into the *Canonical architecture description*.

3.2.3 Canonical architecture description

We cannot expect that all possible components which might cooperate with our component will refer its interfaces by the same names. Moreover these names would have to be the same as author of the component used in the *behavior protocol* provided with it. Even if the author of two components is one person, it is often necessary to have different names for the same binding. For example, first component's input is the second component's output and so on.



Console: `(!out.newLine;?in.putLine)*`
Server: `(?in.newLine;!out.putLine)*`

Figure 3.2: Interfaces with different names are bound together.

A *canonical architecture description* is a *semantically correct description* that uses special interface naming convention. The naming convention is quite straightforward. Its main purpose is to keep information about component bindings. The general idea is to replace original interface names in the protocols by the strings that uniquely identify particular bindings. The string used to replace original interface names contains names of both interfaces involved in the binding. It also contains names of the components to distinguish equally named interfaces from different components.

We already used example from Fig 3.2 to explain consent operator — Fig.2.4. This time the interfaces of the Console component are mutually renamed. It states better the component's notion. To get binding information into protocol of component `Console` we change it to

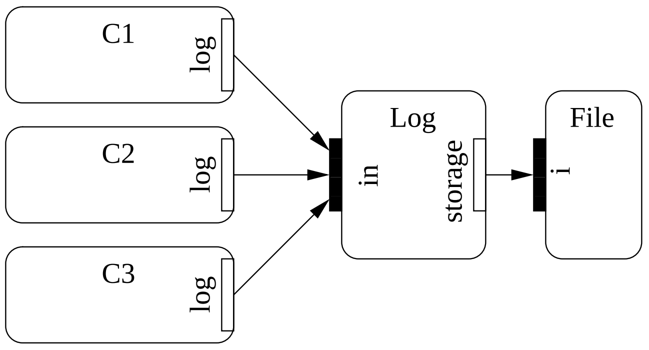
```
(!Console_out->Server_in.newLine;
 ?Server_out->Console_in.putLine )*
```

The first part of interface name is before arrow (→) sign. It is fully qualified and determines *required interface* involved in the binding. The second part determines *provided interface* involved in the binding. The protocol of `Server` component is transformed into

```
(?Console_out->Server_in->newLine;
 !Server_out->Console_in.putLine  )*
```

Multiple bindings

This simple approach works until there are no multiple bindings. Consider example from Fig. 3.3. It depicts an application where three components use services of one logging component. The `Log` component is not able to



Log: `(?in.newMsg{!storage.writeLine})*`

Figure 3.3: Example of multiple bindings.

process requests simultaneously. Using previous simple interface renaming rule, we get three candidates for new name for `in` interface. `C1_log->Log_in`, `C2_log->Log_in` and `C3_log->Log_in`. So we just put an invocation of each method from the multiply bound interface three times.

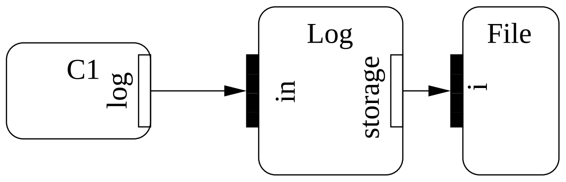
```
(
 ?C1_log->Log_in.newMsg{!Log_storage->File_i.writeLine}+
 ?C2_log->Log_in.newMsg{!Log_storage->File_i.writeLine}+
 ?C3_log->Log_in.newMsg{!Log_storage->File_i.writeLine}
)*
```

The invocations are connected with alternative operator so that the `Log` can still process just one request in each moment. Now, we are sure that all responses are always received by the component that emitted a request.

Multiple threads

The last problem mentioned in the previous paragraph was invocation of one method from two threads running simultaneously within one component. The semantics of the *consent operator* does not ensure that the method response is accepted by the same thread that emitted the request.

We can distinguish threads by adding their numbers to interface names. The Fig 3.4 shows three components. The C1 component can invoke `newMsg` method twice simultaneously. So, the invocations are distinguished by thread number. There must be a special version of complementary event for each thread containing the right number.



C1:`!log.newMsg||!log.newMsg`
Log:`?in.newMsg*`

Figure 3.4: Example of multiple threads within one component.

```
C1:   !C1_log->Log_in#1.newMsg || !C1_log->Log_in#2.newMsg
Log:  (?C1_log->Log_in#1.newMsg + ?C1_log->Log_in#2.newMsg)*
```

As the Log component can process only one invocation, the distinct versions are connected by the alternative operator. This architecture still contains the *bad activity* composition error, but it is caused just by the fact that the components are really not compatible and not by inexact representation of the architecture.

We can use this approach, because the *behavior protocols* are not capable to capture dynamic thread creation. We always know the number of threads the component uses in every state.

An important property of this naming convention is the existence of a simple algorithm that converts a *semantically correct architecture description* into a *canonical architecture description.* This kind of preprocessing greatly simplifies the task of the checking. We need not care about threads and multiple bindings during the checking itself. Also, the information about bindings that the checker requires is limited to mere list of synchronization events.

Under the presumption of absence of multiple bindings, we can also state important property of the *consent operator* — the result of composition of deterministic protocols is also deterministic. This property simplifies the checker too.

Because it requires the information about bindings, this preprocesing is closer to the ADL parser than to the checker, so it is not a part of the distributed checker implementation.

3.3 Consent FSM

Definition 3 *The consent FSM* $CFSM_{A,B}^{Sync}$ *is FSM that accepts all sequences of events that are generated by the protocol* $A \bigtriangledown^{Sync} B$*.*

Definition 4 *The consent FSM* $CFSM^{Sync_1,Sync_2,\ldots,Sync_{N-1}}_{P_1,P_2,\ldots,P_N}$ *is FSM that accepts all sequences of events that are generated by protocol* $P_1 \triangledown^{Sync_1} (P2 \triangledown^{Sync_2} \ldots (P_{N-1} \triangledown^{Sync_{N-1}} P_N)$.

Definition 5 *Let "*P_1 $Sync_1$ P_2 $Sync_2 P_3 \ldots P_{N-1}$ $Sync_{N-1}$ $P_N Unbound$*" is canonical architecture description. Then we say that* $CFSM^{Sync_1,\ldots,Sync_{N-1}}_{P_1,P_2,\ldots,P_N}$ *is architecture FSM.*

The construction of *architecture FSM* is a natural way to enumerate all valid states of the architecture. That's why it is the key part of all current *behavior protocol* checkers. The problem they all have to face is its size.

In order to prove absence of composition errors during the construction of $CFSM^{Sync}_{A,B}$, we have to check the following conditions for each state S:

- Bad activity — Let S_A is a state of $PFSM_A$ corresponding to the S and S_B is state of $PFSM_B$ corresponding to S. For each edge labeled by the synchronized emit event coming from the S_A, resp. S_B there must be an edge coming from S_B, resp. S_A labeled with complement accept event. There cannot be outgoing edge labeled by an emit event from the list of unbound interfaces.
- No activity — State S contains an outgoing edge or S_A and S_B are final.
- Infinite activity — The state S is not a member of a cycle or there is a way from the S to the final state.

3.3.1 Previous versions of the checker

This work is the third attempt to implement *behavior protocol* checker. The first one was based on explicit creation of *architecture FSM* in memory. This approach does not even try to cope state space explosion problem, so its usage is limited to the simple examples far from the practical usage. However, it was the first working one and gave feedback valuable for the next versions of the checker and even development of the *behavior protocols* semantics.

The next version of the checker is based on *Parse Tree Automaton - PTA* [2]. This time, no explicit *FSM* representation is used. The states of an *architecture FSM* are identified only by positions in the parse trees of the protocols. Having a position for each protocol the architecture consists from, we can relatively easily compute the positions in the next step. This way, we can traverse the whole state space. All the visited states are stored to ensure program termination and to prevent repetitive generation of already visited parts. The number of states is still exponential, but this time, only bit-efficient encoded state identifiers based on the positions in parse trees are stored. There is no need to store edges because we can compute them from the parse trees whenever we want.

This approach utilizes available memory much better, so this version of the checker is already capable to check relatively complex architectures. The disadvantage is that the operations with identifiers are time consuming because they are quite complicated. As the protocols are not deterministic in general, not only one position but the set of potential positions for each parse tree must be encoded into state identifier. This yields variable length of identifiers. Another consequence of the complicated state identifiers is rather small set of available operations with them. State neighbours cannot be generated from bit-efficient form of identifier so that another, not so efficient, representation must be used on the stack.

3.3.2 State representation

The state representation used by the distributed behavior protocol checker is somewhere between previous two approaches. It is based on the following presumption.

Presumption 1 *The state explosion is caused mainly by the composition of protocols while the state spaces of particular protocols are relatively small.*

This presumption obviously does not hold in all cases because of parallel operators. Important fact is whether it holds in 'practical' cases.

$CFSM_{A,B}^{Sync}$ is similar to the *cartesian product* of the automatons $PFSM_A$ and $PFSM_B$. The difference is in edges. All edges labeled by events used to model methods from *Sync* are removed. Edges representing τ events are added instead. In other words, one edge labeled by τ event is replacing a pair of edges labeled with complementary events. The resultant automaton is equal to the *cartesian product* if *Sync* is empty. A simple example is on Fig. 2.2.

The distributed checker uses representation of the *architecture FSM* states called *index vectors*. Explicit representation of deterministic *protocol FSM* is built in the memory for each behavior protocol. The states of *protocol FSM* are identified by numbers. The numbers are given by depth first traversal ordering. A state of *architecture FSM* is then identified as a vector of identifiers of *protocol FSM*s' states.

An example on the Fig 3.5 shows *consent FSM* of the protocol `!a*` $\bigtriangledown^{a,b}$ `(?a+?b)*`. Because we only have two *protocol FSM*s, the identifier of *consent FSM* has two parts. The first one identifies a state in *protocol FSM* for

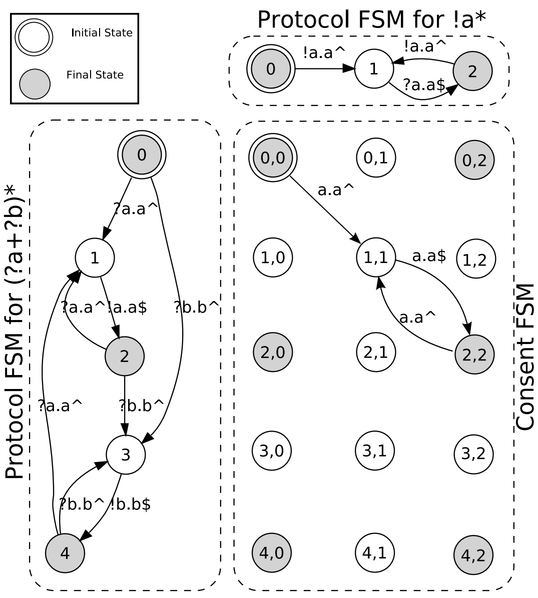


Figure 3.5: State identifiers example

protocol `!a*` and the second identifies a state in *protocol FSM* of `(?a+?b)*`. Note that many states are not reachable.

This representation is similar to the state identification used by Spin model checker [4]. It uses the same presumption that the state explosion is caused by the parallel composition. A state of the Spin model is also identified by a vector of indexes. The n-th index identifies a record in the array associated with the n-th process. There is an array associated with each process that contains descriptions of all reachable states of the particular process. The descriptions consist of states of local variables and a program counter. Because the number of reachable states for each process and even number of processes is not known in advance, the size of the Spin identifier is variable. In our case, we can afford to assign identifiers to all states of *protocol FSM*. Their state spaces are much smaller — there are no variables, just program counter. The number of protocols also does not change in our case. It means that we have a fixed size of identifiers.

In comparison to *PTA based* representation, this one is much simpler. In consequence, the implementation of the operations is simpler too. The most important operations are computation of neighbours and bi-directional transformation of state identifier into bit-efficient version and back.

Having a state identifier $[S_A, S_B]$ of a state from $CFSM_{A,B}^{Sync}$, where S_A is the identifier of a state from $PFSM_A$ and S_B is the identifier of a state from $PFSM_B$, we can easily get all neighbours in the following way:

Algorithm 1 *This function takes state identifier* ***S*** *and the index of the last edge used in each protocol. These three parameters are encapsulated into iterator, which can be then used in a standard way. The function also requires* ***PFSM_A*** *and* ***PFSM_B*** *which are passed in another way.*

It returns state identifier - vector in square brackets.

```
function getNextNeighbour(S,lastEdgeFromA, lastEdgeFromB):

S_A = part of the S identifying state of protocol A
S_B = part of the S identifying state of protocol B


if (hasNextEdge(PFSM_A,S_A,lastEdgeFromA))
    edge_A = getNextEdge(PFSM_A,S_A,lastEdgeFromA)
    if (edge_A in Unbound){
        if( edge_A is emit)
            Bad Activity Detected
        else
            return getNextNeighbour(S,edge_A,lastEdgeFromB)
    if (edge_A in Sync)
        if (edge_A is emit)
            acceptLabel = getComplementEvent(edge_A.label)
            edge_B = getEdgeLabeledBy(PFSM_B,S_B,acceptLabel)
            if (edge_B is Null)
                Bad Activity Detected
            lastEdgeFromA = edge_A
            return [edge_A.target,edge_B.target]
        else
            return getNextNeighbour(S,edge_A,lastEdgeFromB)
    else
        return [edge_A.target,S_B]

else if (hasNextEdge(PFSM_B,S_B,lastEdgeFromB))

    .. the same with A replaced by B and vice versa.

else return Null
```

The situation for more than two *protocol FSMs* is similar. The *bad activity* detection is secured by the algorithm itself. *No activity* is found if a state does not have any outgoing edge and it is not a final one. The *infinite activity* detection is not so easy, so it is explained later.

The transformation into the bit-efficient form is easily expressible by the following formula.

Let $Id = [id_i]_0^{dim-1}$ is an *architecture FSM* state identifier. dim is the number of protocols in the architecture description, id_i is the identifier of the state of i-th *protocol FSM* and $States_{P_j}$ is the number of states of j-th *protocol FSM.*

Then, the bit-efficient representation Id_{eff} is computed using following formulas:

$$Id_{eff} = \sum_{i=0}^{dim} id_i * base_i \tag{3.1}$$

$$base_i = \prod_{j=0}^{i-1} States_{P_j} \tag{3.2}$$

The $base_i$ depends only on the number of states of *protocol FSM*s, so it is computed only once. We have a simple and quickly computable formula for bit-efficient state encoding. Also, the inversion is not very demanding. This is important especially for the distributed environment. We can use only bit-efficient state identifiers for communication between nodes.

Another important property of the *index vectors* representation is that possible non-determinism is eliminated during the construction of *protocol FSM*s . This means that we need not solve non-determinism during generation of *consent FSM.*

In comparison to the *PTA based* representation, this representation has one big disadvantage. It does not address state space explosion caused by parallel operators within one protocol. During creation of explicit *protocol FSM*, we can run out of the available memory easily.

An obvious solution of this problem is to use the same idea one level lower. Use explicit automatons only under the parallel operators. But there are two issues. The parallel operators can occur not only on the top level of parse trees. Consider an example `?i.initialize; (?i.m1|?i.m2)*; ?i.terminate`. It means that number of component threads differs during the execution and that there must be a support for thread creation and stopping. There are two solutions for doing this — *hierarchical FSM*s or support for synchronization.

Synchronization support solves this problem by moving the parallel operator to the top level and postponing its run. Our example would be changed to

```
(?i.initialize; [!P11.start, !P12.start];
                [?P11.stop,  ?P12.stop ];?i.terminate)
| (?P1.start;(?i.m1)*;!P11.stop)
| (?P1.start;(?i.m2)*;!P12.stop)}
```

The synchronization is needed to ensure both threads are terminated atomically. All events from the list in the brackets must be accepted atomically.

The main idea of *hierarchical FSM* is representation of the whole parallel operator subtree by just one huge state. This state contains two independently running automatons — one for each operand. Outgoing edges of the state can be used only if the inner automatons are both in a final state. The inner FSM can be hierarchical again.

Both solutions of this issue would not be easy to implement and it would decrease the speed of the checking. However, the reason why they are not used in the checker is that they do not solve the second, more important problem. It is the non-determinism. The operands of the parallel operator can be non-deterministic. It would not be a problem — we can convert them into deterministic ones. The next source of non-determinism is parallel run of the operands. This kind of non-determinism does not matter too, because in *canonical architecture description* usage of the same method in simultaneously running threads is not allowed for another reason already. The problem is that the non-determinism can be introduced by operators on the higher levels of the parse tree. For example, protocol `A+(A|B)+B` (or-parallel operator) introduces non-determinism.

I do not see any other solution to efficiently cope this non-determinism than postponing it to the *architecture FSM* generation. It would mean encoding sets of states into *architecture FSM* states identifiers. Doing this, we would get much closer to the *PTA based* representation and lost simplicity of *state vectors.*

This is why we build explicit automaton even for parallel operators instead of using on the fly generation.

3.4 State space division

In order to run the task in parallel we must provide rules that split the work among available machines.

The requirements for such division follow:

- Minimal communication between machines — communication is typically much slower than computing itself.

- The amount of work assigned to particular machines must be comparable — for best utilization of available resources.
- Minimize duplicity work done by distinct machines

These requirements are not orthogonal and influence each other.

Let an architecture description contains n behavior protocols. Then the proposed state identifiers are vectors having n elements. As the number of states is finite, we can enclose all state identifiers into n-dimensional cuboid. Thanks to the depth first traversal based ordering of states from *protocol FSM*, we can be sure that if two vectors are close, the states they represent are close too.

If we use *index vectors* as representation, we can formulate problem of state space division as assigning parts of n-dimensional cuboid to the available machines. We say that an edge is *crossing machine boundary* if it heads from the area assigned to one machine to the area assigned to another machine. The number of states assigned to one machine is number of states that belong to one of the areas assigned to the machine.

The good assignment has small number of edges crossing machine boundaries and each machine has similar number of assigned states. The small number of edges crossing machine boundaries means small communication between machines. By assigning a similar number of states to all machines we achieve uniform workload distribution. Using explicit assignment of the states to machine we ensure that no state is examined by two distinct machines. Thus, there is no duplicity work caused by distribution.

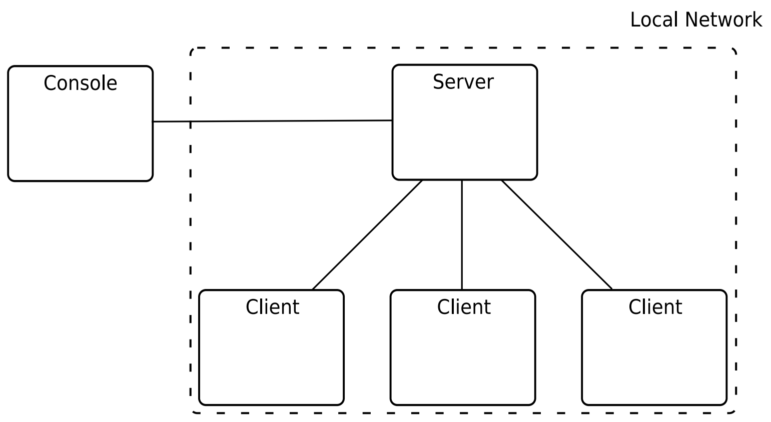


Figure 3.6: Participants

Before continuing with distributed traversal algorithm, I will present participating processes. See Fig. 3.6. The reason for using a distributed approach is not to achieve higher robustness in our case. So, we can afford to have a special and unique 'arbitrator' process called *server*. A user communicates with the *server* using the *console* process. The work itself is done by *client* processes.

When the user wants to check a protocol, he uses the *console* to pass the *canonical architecture description* to the *server* as a string. The *server* creates a parse tree and *protocol FSM* for each protocol from the description. At this moment, problems with insufficient memory caused by many parallel operators can appear. When the automatons are constructed, they are sent to the *clients* and the checking can begin.

Algorithm 2

```
Divide the cuboid among the machines
Start the traversal of the area containing the initial state
If the machine A achieves a state S out of its areas
    send the state S to the server
The server decides that the state S belongs to the machine B
The machine B obtains the state S from the server
If B was idle it starts traversal from the state S
else B adds the state to the queue

Finish when all machines are idle and all messages were
received.
```

This simple algorithm does not provide the answer to two important questions. How to divide the cuboid among machines and what traversal strategy should be used.

3.4.1 Traversal strategy

There are two commonly used traversal strategies. Breadth First Search — BFS and Depth First Search — DFS. DFS is used more often in model checking. Its advantage is that when an error is detected, a counter example is already on the stack. However, in the distributed case, we have more stacks. When one machine reaches its boundary, the state identifier is sent to another machine. But the first one does not wait for the result of traversal performed by the second machine. It means that we cannot exploit this feature for reporting errors. DFS used for traversal of *architecture FSM* is kind of local because the identifiers of the *protocol FSM* states are assigned using DFS too. It means that it begins with the states identified by vectors containing low indexes and smoothly continues to higher indexes.

The memory requirements for the stack are given by the size of the longest trace. The size of the queue used in BFS is not limited so naturally. In the worst case it can be exponential to the size of the longest trace but it is very rare.

The experiments shows us, that the memory requirements does not give us strong preference.

An important difference is that the BFS traversal is not local. It quickly skips to the vectors with higher indexes. This locality property shows to be important for final decision.

3.4.2 Dividing the cuboid

Static assignment

The idea of the static assignment is simple. Let the number of available machines is m. We know in advance how *protocol FSM*s looks like. We can use this knowledge for splitting the whole cuboid into m areas. The traversal is started by the machine that obtained the area containing the initial state. As we want the rest of machines to start immediately we use BFS traversal strategy. It ensures that the vectors that belong to the other areas are reached pretty soon. Fig. 3.7 shows an example.

In this example, the simplest division was used. The cube is just split to two halves of equal volume with a hope that the number of reachable states within these areas is similar. This division utterly ignores the second criterion — the number of edges crossing the machine boundaries.

The disadvantage of the static assignment is obvious — if the initial division is bad, we can not change it later. And we do not know that particular division is bad until we try it. It can happen that some area is even inaccessible. In such case, the machine this area was assigned to is idle during the computation and its memory remains unused.

However, for many cases, this solution works better than others, because it has the smallest overhead. Also, we know in advance which part of *protocol FSM* particular *client* will need. Thus, we can send only the necessary parts to each machine. It can be very useful especially in cases where Presumption 1 does not hold at all.

Dynamic assignment

This approach is similar to the previous one. At the beginning, we have the same n-dimensional cuboid and m machines. This time we split the cuboid to more parts than m — let us say q. Only the area containing the initial

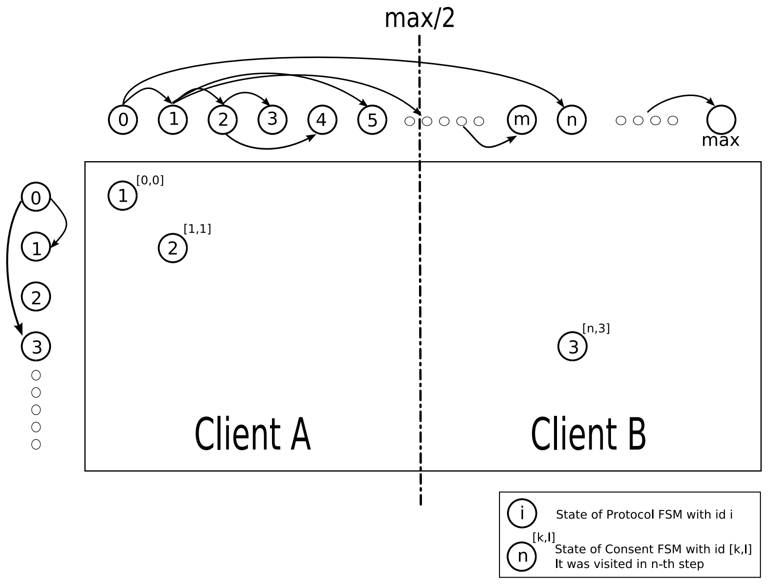

max/2
0
1
2
3
4
5
m
n
max
[0,0]
[1,1]
[n,3]
Client A
Client B
State of Protocol FSM with id i
[k,l]
State of Consent FSM with id [k,l]
It was visited in n-th step

Figure 3.7: Static assignment — Let us have two machines and an architecture description containing two protocols. Along the edges of the cuboid are *protocol FSMs*. The states of *architecture FSM* are within the cuboid. We know the number of states of *protocol FSMs* so we can split the cuboid into two parts and assign them to the machines. The area containing the initial state is assigned to the machine A, so it can start the traversal. The machine B is waiting until the machine A reaches a state from the second area. Because the BFS traversal is used, the state from second area is reached in the third step.

state is assigned to one of the machines. Then, traversal of the initial area is run. Again, BFS is used to reach a state out of the initial area. When the *server* obtains a state identifier, which is a member of an area, which was not assigned yet, it assigns the area to one of the idle nodes. If no machine is idle, the *server* must remember the state identifier until such machine appears.

Once an area is assigned to a machine the assignment cannot be changed. The problem is that one area can contain more parts of *architecture FSM*, which are not connected within area. In consequence the machine can become iddle although there are still many states reachable from the initial state that were not visited. The iddle machine asks for another area. This way number of areas containing majority of reachable states can be assigned to one machine.

Number of areas q is twofold. The more areas we have, the better approximation of reachable states we get. It means uniform utilization of available memory and CPU power. On the other side, more areas mean more boundaries and more communication and overhead.

There is also a requirement to the *server* process to remember the reached states of the unassigned areas. There can be a problem with insufficient memory if wrong division of the cuboid is chosen.

Dynamic reassignment

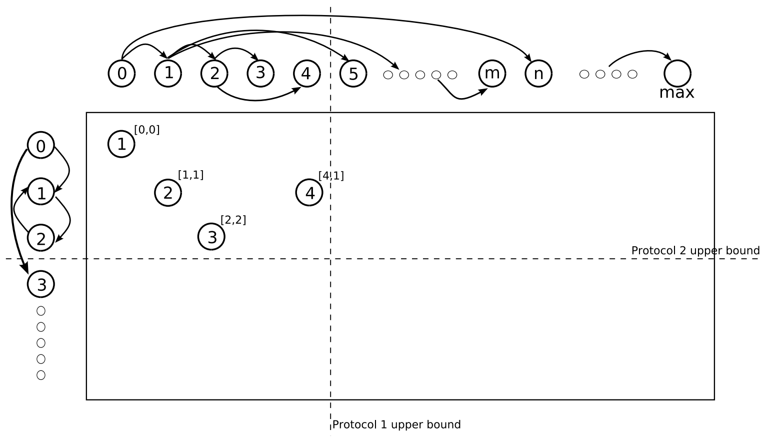
While the previous two solutions were rather similar, dynamic reassignment is completely different. It starts with assigning the whole cuboid to a machine. This machine starts the traversal from the initial state. If there is an idle *client B*, other *client A*, which is not idle, is asked for part of its area. The *client A* then interrupts its own traversal and splits the assigned area into two parts. It is important to split the area in such a way that one part contains all states already reached by *client A*, while the rest is completely untouched. This is achieved by using DFS traversal strategy.

Here, we are utilizing the locality of DFS already mentioned in the section about DFS. The bounding cuboid of all reached states is obtained by remembering the maximal identifier used in each *protocol FSM*. When a *client* splits one of its areas, at least one state from the new part must be reached quickly. We need to have an entry point reachable from the initial state. To get such state soon, BFS strategy is used for a while.

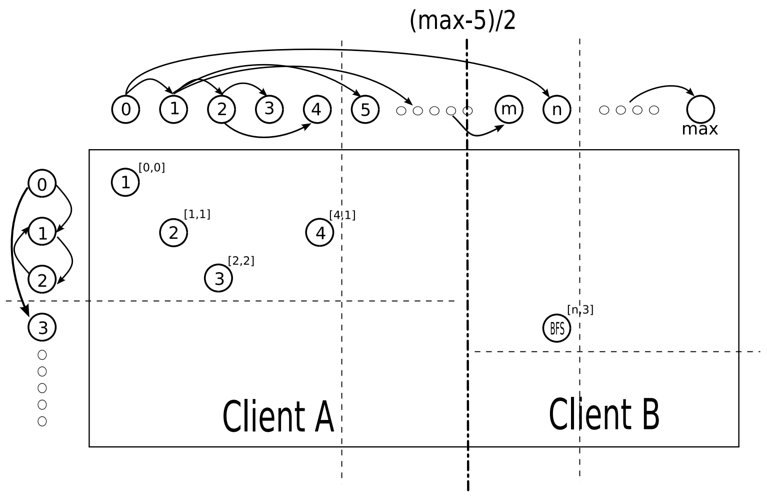
The example on Fig. 3.8 we already used. This time, *dynamic reassignment* is used. Fig. 3.8 (a) shows a situation shortly after beginning. There is only one running *client A* performing the DFS traversal. Notice the upper bounds. Each visited state is in a bounded area. This is important, because we are sure that when we create a new area within the rest, no visited state is here. In consequence, set of the visited states within the new area is empty. Thus, we need not transport it to the new destination.

After a while, a request from the *client B* appears. The *client A* stops the DFS traversal for a while and somehow splits the untouched area behind bounds into two parts. Then, the *client A* must reach at least one state from the new area which is reachable from initial state. To obtain it quickly, BFS is used for a while. The BFS queue is initialized with the actual DFS stack. When a state belonging to the new area is reached, the area is delegated to the *client B*, which starts its own traversal. Each area manages its own upper bound. The beginning of the *client B* traversal is on the 3.8 (b).

This approach is the most complicated and therefore it has the biggest overhead. Let us focus to an unpleasant consequence. The *client B* continues with the traversal. It also uses DFS, so initially the vectors containing rather low states are examined. It means that the *client B* often touches the



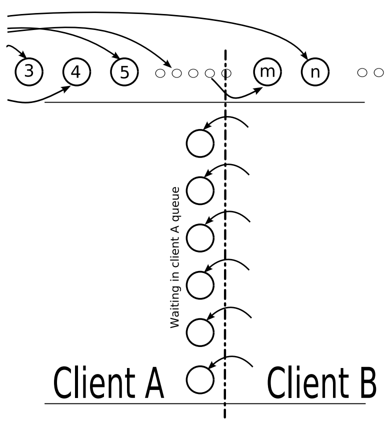
(a) The situation shortly after beginning. The entire cuboid is assigned to the client A, which is performing DFS.



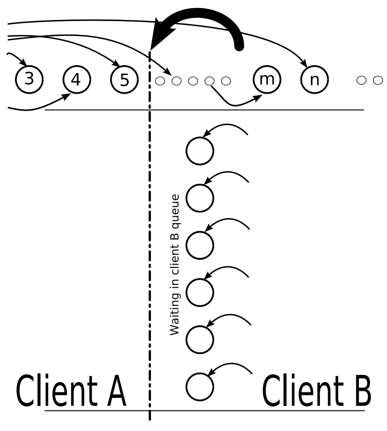
(b) The client A split its area, which was assigned to the client B.

Figure 3.8: Dynamic Reassignment

boundary. The states belonging to the *client A* are transferred. But if the *client B* becomes idle before the *client A* it asks the *client A* for the next area. The *client A* again assigns the right part of its area. But there is lot of states waiting in its incoming queue that belongs to the newly assigned area. So they all must be transferred back to the *client B*. An example is on Fig. 3.9.



(a) States are collected in the client A queue.



(b) All states must be moved back to the client B.

Figure 3.9: Dynamic reassignment overhead example

Because the assignment can change in each moment, we must really use *server* process for routing the states to their destinations.

3.4.3 Dividing the state space in the local checker

Division of state space can be useful even in the local version of the checker. The advantage appears when we are checking a task which does not fit into the physical memory and the model checker is slowed down by swapping.

Consider a checker using one huge hash map, which does not fit into the physical memory, to remember identifiers of all reached states. When neighbours of a state are reached, the hash function typically assigns to them very different numbers. It is exactly the way a good hash function should behave. It also means that with a high probability the virtual memory addresses used to store each neighbour is far from the other addresses and also from the address where the parent's identifier is stored. But the virtual memory performance strongly depends on the locality of memory accesses.

To use the locality hidden in the architecture, all we must to do is to divide the state space. Each part owns its own hash map used to store reached states from this part. These local hash maps contain less states so they fit into the physical memory. When one part of the state space is examined, all neighbours of a state belongs to the same hash map with a high probability. And if we are checking this part of the state space for a while, it is already loaded into memory.

3.5 Reporting errors

The *PTA* based checker provides two different ways of error reporting. The first one is the *error trace* — a sequence of events labeling the path from the initial state to the state where a composition error was detected. The second one is the *annotated protocol* — input behavior protocol containing special marks identifying positions where the error was discovered.

Using the sequential DFS is the simplest way to get the *error trace*. There is only one stack containing the path from the initial state to the state being currently examined. All that must be done is to print out the labels along this path if composition error is detected.

Because we do not use only DFS and even DFS is run in parallel, we have many stacks. Even the data from the different stacks does not typically create a path as the distinct traversals do not wait for each other.

We need an additional data structure for reporting a stack trace. It allows us to provide *error traces* independently on the chosen traversal strategy. We already have a structure for remembering the visited states. Here, except for the state identifier itself, we can also remember the first state each item was reached from. Doing this, we are getting the *spanning tree* of the graph representing the automaton. When a composition error is found, a path coming from the initial state to the current state can be reconstructed from this structure.

In order to report the *annotated protocol*, we must keep an mapping of *protocol FSM*s states to the parse tree nodes.

3.6 Infinite activity detection

Detection of the *infinite activity* is more complex than the detection of the other composition errors. It requires information gathered from many states. These states can be distributed over all machines. That's why we are going to discuss it after the section about distribution.

The architecture contains *infinite activity* if and only if there is a cycle of states reachable from the initial state such that there is no path from the cycle to a final state.

There are two relatively independent problems that must be solved — managing cycles and detecting whether a final state can be reached from a particular state.

3.6.1 Reachability of final state

A simpler task is deciding whether there is a path coming from a particular state to a final state. Using sequential DFS, we can get it quite easily. When returning from a subtree, we know about each child whether there is a path from the child to a final state. So we can say that there is a path from the particular state if the same property holds at least for one of its children. But this is not our case because we do not use single-threaded DFS. To get this information, we must use the *spanning tree*. It is created during the traversal and remains in the memory when the traversal finishes.

Algorithm 3 *Visited is spanning forest containing visited states. Each state remembers one of its parents. OK is a set containing states a final state can be reached from.*

```
for each final state FS from Visited
    move all states on the path form initial state to FS to OK
while (Visited is changing )
    for each leaf S from Visited
        for each C child of S (architecture FSM)
        if C is member of OK
            move S and all ascendants from Visited to OK
            break
```

Remember that *spanning tree* is typically distributed over all clients and that the set OK can be too large to fit into the memory of one computer.

3.6.2 Managing cycles

Now, let us focus on cycles. Instead of providing clever way to manage them, we prove that we need not manage them.

Theorem 1 *Let architecture FSM does not contain the bad activity or no activity composition errors. Then the system contains infinite activity if and* *only if it contains a state w such that there is no path from the state w to a final state.*

By definition, the system contains *infinite activity* if there is an cycle $s_1, s_2, \ldots, s_k$ such that $\forall i \in \{1 \ldots k\}$ no *final state* is reachable from s_i. We can denote one of the states w and implication from left to right is done.

Presumption of the opposite implication is that we have a state w such that no *final state* is reachable from w. Let us take an arbitrary path p starting in the state w and finishing in the state z such that z does not contain any output edge. Let us suppose, that the path p is not a prefix of another path. There can not be a *final state* on the path p because it would be contradiction. So the state z is not final, but it does not have any output edge. It means that there is *no activity* or *bad activity* in the system, which is contradiction with theorem presumption. In consequence all paths starting from the state w are prefixes of other longer paths. Because *architecture FSM* is final each path starting from the state w contains a cycle. And because there is no final state reachable from the state w, there is also no final state reachable from this cycle.

When we are sure that a system does not contain other composition errors, we need not care about cycles. But we are detecting infinite activity from the *spanning tree* after the construction of *architecture FSM*. If there was another composition error, it would be detected in the previous phase during the construction.

3.7 Heuristics for state space division

Until this point, we did not consider different ways of splitting the state space. In the previous examples, we just divided it into few parts of the same size. This splitting respects only the requirement for the similar number of reachable states assigned to the machines. It utterly omits the requirement of minimal number of edges crossing machine boundaries. On the other side, the best splitting from the point of view of minimal number of crossing edges is no splitting. We need to find a balance between these two requirements.

A very coarse estimate of the number of reachable states is volume of the n-dimensional cuboid.

If we want to respect crossing edges, we must use more information from *protocol FSM*s than the number of states.

Let us have the architecture A. Its behavior is described by $CFSM_A = CFSM_{P_1,P_2,\ldots,P_n}^{Sync_1,Sync_2,\ldots,Sync_{n-1}}$. Let $Cube_A$ is a n-dimensional cuboid containing all state identifiers of $CFSM_A$. Let us denote $CE_{P_k}(l)$ as the number of edges crossing the boundary between two areas obtained by splitting $Cube_A$ at the number l on the axis k. $States_{P_i}$ is the number of states in $PFSM_{P_i}$ and $E_{P_k}(l)$ is the number of all edges E in $PFSM_{P_k}$ such that $E =< Id_a, Id_b >$ where Id_a and Id_b are state identifiers of connected states and $Id_a <= l < Id_b$.

$$\forall k, l : CE_{P_k}(l) <= E_{P_k}(l) * \prod_{i=0, i\neq k}^{n} States_{P_i} \tag{3.3}$$

In the following text, the right part of the equation is referred to as *crossing edges upper bound* $CE_{P_k}^{upBound}(l)$

$$CE_{P_k}^{upBound}(l) = E_{P_k}(l) * \prod_{i=0, i\neq k}^{n} States_{P_i} \tag{3.4}$$

We want to chose k and l such that $CE_{P_k}^{upBound}(l)$ is minimal. However, we do not get minimal $CE_{P_k}(l)$ necessarily this way.

Algorithm 4 *Computation of $E_{Prot}(l)$ for a particular protocol FSM. The result is in the array EdgesOver, which is an array of the size equal to the number of states in the protocol FSM. Items are initialized to zeroes. DEdgesOver[i] contains differences between EdgesOver[i] and EdgesOver[i+1]*

```
For each Edge E=<S1,S2> from protocol FSM
    DEdgesOver[S1]++
    DEdgesOver[S2]--

sum = 0
for( i = 0 ; i<number of states ; i++)
    EdgesOver[i]=sum
    sum+=DEdgesOver[i]
```

This way we can split a cuboid with respect to the number of edges crossing machine boundaries. However the volume assigned to particular machines can differ.

Let us suppose that we want split $Cube_A$ along the axis k into p parts. The protocol corresponding to the axis k has $States_{P_k}$ states. It means that we are looking for splitting points $s_0, s_1, \ldots, s_p$ such that $\forall i \in \{0, \ldots, p-1\}$: $s_i < s_{i+1}$, $s_0 = 0$ and $s_p = States_{P_k}$.

If we want to numerically rate how much a particular division comply with the requirement for the similar area volumes, we can use variance of splitting point distances σ^2.

$$\sigma^2 = \frac{\sum_{i=0}^{p-1}(s_{i+1} - s_i - \frac{States_{P_k}}{p})^2}{p} \tag{3.5}$$

Finally, we can combine both criteria into one expression.

$$Unsuitability = C_{edges} * \sum_{i=1}^{p-1} CE_k^{upBound}(s_i) + \sigma^2 \tag{3.6}$$

Where C_{edges} is a constant expressing how many times time takes transmitting of one edge than checking of one state. We are looking for $s_0, s_1, \ldots, s_p$ such that $Unsuitability$ is minimal.

This rating can be easily extended for splitting in more axes.

Finding the smallest unsuitability might not be easy. The methods of *constraint programming — CP* can be used [13]. CP tasks are often NP-complete, but recall that the number of splitting points is typically small, somehow derived from the number of available machines.

It is important to mention that all of this is just a heuristic. By choosing a division that has rather small unsuitability, we avoid 'obviously bad' divisions. Moreover, this heuristic depends on the estimation $CE_{P_k}^{upBound}(l)$ of the number of edges crossing areas. The experiments shows that this upper bound is too coarse. The graphs comparing this estimate with real values are in the appendix C.

To conclude this section, I did not find any heuristic that works well in average case. It is a good topic for future work. The current implementation thus divides the state space to the parts of equal volume.

Chapter 4

Implementation

4.1 Language, platform, libraries

The tool is implemented in Java 1.5. The main reason is that the rest of the Sofa framework including the previous version of the checker is also implemented in Java. The built-in middleware and networking support together with the platform independence were also considered to be useful.

The parser of architecture descriptions is generated using JavaCC — Java Compiler Compiler.

For building, Apache ANT is used.

4.2 Distributed checker processes

There are three different kind of processes used in the checker. They are depicted on Fig 3.6. The *server* process is used to manage information about registered clients and to control the computation. The number of the *client* processes performs the checking itself and finally, the *console* process is used to submit a task to the *server* and to report the result.

4.2.1 The Server

When the *server* process is started, it exports two Java RMI interfaces. One is used to accept requests from the *console* process and the other is used by the *client* processes for registration.

When the user submits a task from the *console* process, the *server* creates a new RMI object representing the new task and notifies all *clients*. This object cares about assigning the parts of the state space to the *clients*, sending the states to the proper machines and termination detection. There are three different task classes provided. Each is specialized to a different assignment strategy. The user chooses the one he wants to use by setting a Java property.

The object architecture description is created during the initialization of the *server* task object. It consists mainly from creation of finite state machines. The *server* task object also opens a socket for each client. These sockets are used for transmitting states.

During the checking, the server is only waiting for messages coming from both — sockets and RMI. From the sockets, new state identifiers are coming. If there is a state identifier coming from a particular *client*, it means that this state was reached by the edge crossing boundary of the area assigned to the client. The server knows where the new state belongs to and writes it into the proper socket. It means that all such states go through the server. This is potential bottleneck. I decided for this solution to support dynamic *reassignment*. This was the simplest way to manage information about the state space division, which is changing. Other assignment strategies — *static* and *dynamic* would allow sending of states directly between clients.

Through RMI interface, the server obtains notifications about the idle *clients* and the composition errors found. If a new idle client notification occurs, the server tries to somehow obtain a new area — the way it is accomplished depends on the chosen assignment strategy. The new area is assigned to an idle machine. If there is no more work, the machine remains idle. If all machines are idle, termination detection is performed. If it succeeds, the task is finished — the *clients* and the *console* are notified and the exported task object is destroyed. If a composition error is found, the task is terminated immediately.

If a communication error occurs, we cannot continue. In such a case, the task is restarted from the beginning using only the rest of available clients.

Two different approaches for communication were chosen, because requirements for sending the state identifiers are different from the rest of transmitted events. Sending many few-bytes-long state identifiers over RMI would cause huge overhead. Using sockets, we can use buffering to transmit many identifiers in one network packet. The rest of communication is done via RMI, because it is easier to implement and maintain.

Notice that there is very little communication between the *console* and the *server*. The *console* can be far from the *server*, while the *clients* should be in the same network segment as the *server*.

4.2.2 Clients

When a *client* process is started, it is registered at the *server* process. After the registration, it waits for the task. When a task is assigned, a new RMI object representing the task from the point of the *client* view is created. Then, the *client* process still does not have its own part of the state space, so it becomes idle and notifies the server about it. The *server* assigns a new area to the *client* in response. If the new area contains at least one entry point, the traversal can begin. The entry point is a state reached by another *client* or the initial state. Again, we have different task objects for different traversal strategies. Client task object for the *static* assignment is the simplest one. There is only one area assigned to each client in this case. It uses the BFS traversal strategy. The same strategy is used also by the task object for the *dynamic* assignment. This time, many areas assigned to each machine must be maintained. The task object specialized for *reassignment* uses the DFS strategy.

There are two types of areas — one for the BFS and other for the DFS traversal strategy. Each area contains a structure that represents the *spanning tree* of the visited states. It is used to remember already visited states and for reconstruction of the error trace. It can be also used for the *infinite activity* checking, but this feature is not supported in the current implementation.

During the traversal the client is receiving the states that were generated by other clients and that belong to one of its area. It is also sending the states that generates and does not belong to any of its areas to the *server*.

4.2.3 Console

The *console* process does not do anything else than reading architecture description, sending it to the server as a string and waiting for response. The intention was to make it as simple as possible so that new consoles can be implemented with a minimal effort. It should help to incorporate the checker into the existing Sofa ADL parser. Another interesting option is implementation of a web interface to the server.

Currently, just a simple command line version is implemented.

4.2.4 Termination detection

The termination detection is performed by the server when all clients are idle. It is based on the counting overall number of states received and sent. When the number of the received states is equal to the number of states sent and all clients are still idle, we can terminate the task. The same approach is used in [14] and [15].

4.2.5 State space division

All three presented approaches were implemented. However, the *reassignment* strategy does not behave well because of big overhead. As the results are definitely not promising the reassignment strategy will not be supported in the next versions. It will allow us to send the states among machines directly, not through server.

4.3 Architecture description

The *server* component obtains an architecture description as a string from the *console*. JavaCC generated parser is used to create the parse trees of all protocols involved in the architecture. The parser also fills the `EventTable`, which holds a translation table between event codes used in the rest of the tool and the textual representation. Each protocol is represented by the `ProtocolKeeper`. This class is used to manage different representations of a protocol.

4.3.1 Parse trees

The parse tree represents the string exactly including all abbreviations and and unnecessary brackets to be able to print it out again. The nodes representing events remember event code assigned by the `EventTable`. Supported operations over the parse tree include creating of the *inverted protocol*, printing the protocol out, printing the annotated version of the protocol and the construction of FSM. They are implemented using the *visitor pattern* [16]. The printing out of the annotated protocol is parameterized by the set of parse tree nodes. The result string just contains marks before the nodes from the given set.

4.3.2 Finite state machines

When the parse tree is created it is transformed into FSM. It can be nondeterministic. The transformation is done in bottom up manner. Creation of FSM representing an elementary behavior protocol consisting of just one event is trivial. A special method exists for each operator. It takes the number of state machines each representing a subtree of one child. Result of this method is FSM corresponding to the whole subtree. The result of the whole operation is also stored in the `ProtocolKeeper`.

This FSM representation still contains references to the parse tree to keep information needed for printing out of annotated protocols. It is also capable to keep non-deterministic FSM. It is not used for accepting of words, but it is used only for the construction of other FSM representation, which is used for distribution to the *clients*. That is why we don't need fast look up among the edges leaving a state. All edges leaving the state can be stored together in a set although they are labeled with different labels.

The FSM representation suitable for distribution is called `SimpleFlyFSM`. It does not support for non-determinism. When the first version of FSM created from the parse tree is already deterministic, we can immediately create a corresponding `SimpleFlyFSM`. If it fails, it means that original FSM is non-deterministic and the transformation into a deterministic one must be performed at first. At this point, the minimal version of automaton could be also created. It could decrease memory requirements.

So, at the end of this stage, we have three or four different representations of the protocol stored in the `ProtocolKeeper`. But, we don't need them all any more. The intermediate representations can be forgotten, we keep just the parse tree and the `SimpleFlyFSM`. To enable the printing out of annotated protocols, we must also remember a function that assigns to each state of `SimpleFlyFSM` a set of nodes from the parse tree it corresponds to. We need a set, because of possible non-determinism hidden in the deterministic states of the `SimpleFlyFSM`. The function is represented by a map from integer to a set of parse tree nodes. If the annotated protocols feature is off, it is not created.

4.3.3 FlyFSM

The `FlyFSM` is a representation of FSM, which does not keep its states explicitly in the memory. It provides methods for state neighbours enumeration and for look up using the edge label, but instead of reading it from the memory it always computes it. It is always implementation of the *consent operator*, which somehow uses the data from explicit FSMs stored in the `SimpleFlyFSM`. When the *server* computes a `SimpleFlyFSM` for each protocol of the architecture, it uses it to construct `FlyFSM`. This `FlyFSM` is then distributed to the *clients*.

SimpleFlyFSM

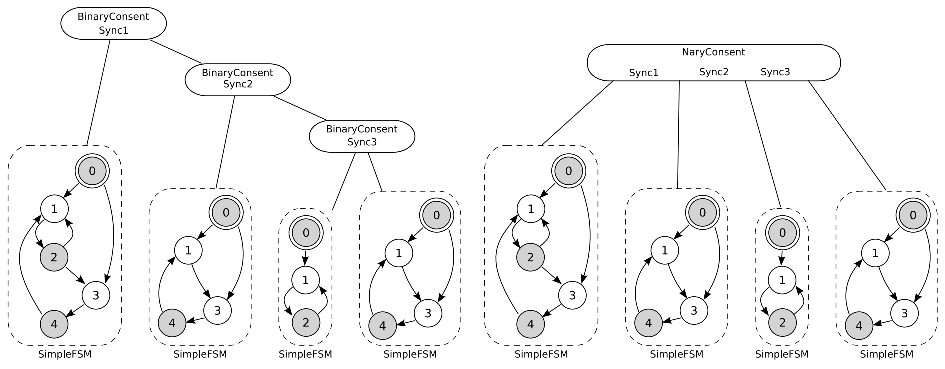
The `SimpleFlyFSM` is a representation of FSM designed for easy distribution by parts using the Java serialization. The states are identified by integer numbers. These identifiers correspond to the DFS ordering. There are two state representations available. They differ in implementation of the output edges. One uses a sparse array indexed by the edge label, while in the other case the binary search in a dense array is used. The first one is fast, the second one is small, and neither supports non-determinism.

The target of an edge is always represented by an integer identifier, not by a Java reference. It enables us to easily split this automaton and send each part to a different machine using the Java serialization.

The consent operator

There are two implementations of the *consent operator* — a binary and a n-ary version. The binary version takes two operands. The first one is always `SimpleFlyFSM` while the second can be also `SimpleFlyFSM` or binary consent. The whole architecture is described by binary tree — see Fig 4.1(a). The binary consent is parametrized by the set of synchronized events.

The n-ary version uses all `SimpleFlyFSM` of the architecture directly. We have two implementations, because the binary version is simpler to implement, while the n-ary version is faster. The binary version is kept because it can be modified much easily in order to implement new features in future.



(a) Binary consent (b) n-ary consent

Figure 4.1:

State identifiers

A state of FlyFSM is identified by a vector of integers as described in 3.3.2. However, the memory efficient representation from the equation 3.1 is usable as long as the Id_{eff} fits into 64 bits — the Java long type. If it does not fit, we have to use a big integer implementation. But the inversion of the original formula contains division and modulo. These operations take a lot of time using software implementation of big integers. To avoid this, all multiplications in the formula are replaced by bit shifting. In consequence, we can multiply (or divide) only by powers of two. We can use a bigger $base_i$ than necessary. Doing this, we get a representation, which is slightly less effective. At worst case we can lose one bit for each protocol used in the architecture. However, this implementation is as fast as a naive implementation using long and much faster than a naive implementation using Java BigInteger.

Chapter 5

Evaluation

This chapter presents results achieved by implementation described in the previous chapter. The evaluation has two parts. First, we compare local version of the checker with *PTA* based checker. Then, the scalability of the distributed version is presented.

5.1 The architectures used for evaluation

5.1.1 Artificial

This architecture contains few protocols including only the sequential operator and the parallel operator. Parallel operators are always on the top level. Because the architecture is quite simple, we can smoothly change the amount of parallelism represented by the parallel operator and the consent operator.

Such an architecture contains many edges and can be seen as the worst case of the task from communication overhead point of view. On the other side, we can slightly modify the architecture using the alternative operator on the top most level to split the state space into two parts of equal sizes. Such a state space can be divided among two machines with only one edge crossing the machine boundary. This can be seen as the best case.

The preprocessing stage of the artifical architectures is very short so it does not affect the scaling results.

5.1.2 Real-world

We are really interested in protocols describing 'real' applications. Such a set of protocols was already developed by DSGR at Charles University as a case study of behavior protocols [17],[18]. I took the architecture descriptions from this project, removed atomic actions, which are not supported by the distributed checker and changed them a little bit to comply with the grammar of the distributed checker input file.

5.2 The local version vs. PTA

Here, we compare the local version of the distributed checker with the previous PTA based checker. We use only the protocols from the case study. They can be found on the thesis CD.

Architecture	States	Time[s]	Speed[states/s]
FreqDB_FlyTicketdb	15	0,25	60
token	24	0,39	61,86
flyticketdatabase	32	0,49	65,71
arbitrator_dchp_frames	1080	0,82	1315,47
Dhcp_other_frames	3456	2,44	1414,08
ipaddressmanager_transientipdb	71969	64,56	1114,83
ipaddressmanager_timer	654350	399,86	1636,45
cardcenter_accountdb_token	131798	124,81	1055,96
arbitrator	1703125	758,8	2244,51
toplevel_frames	1220856	1111,65	1098,24
arbitrator_vs_frame	7822302	8633,12	906,08

Table 5.1: Results of the PTA-based checker

Architecture	States	Time[s]	Speed[states/s]
FreqDB_FlyTicketdb	14	0,51	27,34
token	24	1,62	14,86
flyticketdatabase	51	0,3	170,57
arbitrator_dchp_frames	1683	1,23	1366,07
Dhcp_other_frames	3069	1,5	2052,84
ipaddressmanager_transientipdb	7456	0,68	10900,58
ipaddressmanager_timer	26613	1,32	20207,29
cardcenter_accountdb_token	179715	5,84	30799,49
arbitrator	1000000	64,98	15388,88
toplevel_frames	1903278	92,8	20510,57
arbitrator_vs_frame	4096170	128,75	31814,91

Table 5.2: Results of the local version of the distributed checker

The distributed checker was run with the default settings. The *infinite activity* detection of the PTA-based checker was turned off as the distributed checker does not support it. The tests were run on a computer containing AMD Opteron(tm) Processor 144 on 1800 MHz and 1 GB RAM.

From the results, we can see that the PTA-based checker is much slower. The distributed checker is significantly faster even without distribution.

Another interesting observation is that the same architecture is typically modeled by a smaller automaton in case of distributed checker. The reason is that both checkers uses a different translation from the parse tree of regular expression to FSM. You can see examples of translation of elementary protocols in the Appendix A. This translation influences the resulting number of states a lot. Another reason of the difference can be hidden in the complicated function used to create the bit-efficient representation in the PTA-based checker.

5.3 Scaling

There are only three architecture descriptions large enough for distribution in the case study. We add two 'artificial' architectures — big.bp and big5.bp. The first one is a parallel composition of sequentially connected events and the other one contains two parallel compositions connected by the alternative operator. The input files are in the `testfiles` directory on the CD accompanying the thesis.

All tests were run on the same computers as the local version in the previous section. The measured times are the average values from four attempts. The number of crossing edges is taken from the run which is closest to the average time.

	Static		Dynamic		Reassign	
Clients	Time [s]	CE[1]	Time [s]	CE	Time [s]	CE
1	73	0	72	21	73	0
2	41	413036	46	842673	70	404271
3	31	808496	42	1650099	69	769915
4	26	1195168	59	2414669	66	924389
5	30	1584037	67	2918980	93	796104

Table 5.3: Results for big.bp — 4826809 states

[1]Edges crossing machine boundary

In table 5.3 you can see the results for the big.bp architecture. This kind of task is not suitable for dynamic assignment. It is obvious, because all states of cartesian product are reachable in this case. Dynamic assignment helps us to better approximate reachable areas, but it is not needed in this case. It just increases the number of crossing edges. The same arguments hold for the reassignment.

	Static		Dynamic		Reassign	
Clients	Time [s]	CE	Time [s]	CE	Time [s]	CE
1	152	0	147	99	141	0
2	71	8788	76	781474	78	208443
3	52	830466	57	1544727	126	482913
4	39	834860	62	2340955	132	648320
5	35	1619189	70	2434665	131	465561

Table 5.4: Results for big5.bp — 9660209 states

The table 5.4 contains results for the architecture description, which can be staticaly distributed among odd number of machines very well. The reason is that its state space consist of two big parts connected just by one edge.

Artificial architectures have the advantage that they comply with the presumption about sizes of protocol automatons. The splitting of state space to the parts of equal volume brings uniform distribution of reachable states to the machines.

Now, let us move to the architectures from the case study. In these cases the server side construction of protocol automatons becomes relevant part of overall time required for the checking.

	Static		Dynamic		Reassign	
Clients	Time [s]	CE	Time [s]	CE	Time [s]	CE
1	70	0	62	88	61	0
2	56	213864	63	384504	85	443885
3	54	402073	59	584701	107	528896
4	59	553297	71	714216	123	580776
5	60	626594	66	742194	140	453071

Table 5.5: Results for arbitrator.bp — The whole automaton has 1000000 states. The times include the creation of protocol automatons by the server. It takes 26 seconds in this case.

The architecture description `arbitrator.bp` definitely does not comply our presumption. It contains one huge protocol translated into an automaton with over 700000 states. It is combined with a tinny protocol represented by 5 states automaton.

	Static		Dynamic		Reassign	
Clients	Time [s]	CE	Time [s]	CE	Time [s]	CE
1	96	0	88	0	86	0
2	81	255999	92	182333	137	90981
3	72	236535	81	627584	171	118003
4	72	445755	101	911379	196	116416
5	72	370471	100	845478	220	114829

Table 5.6: Results for toplevel_frames.bp — 1903278 states. The protocol automatons creation takes 35 seconds.

	Static		Dynamic		Reassign	
Clients	Time [s]	CE	Time [s]	CE	Time [s]	CE
1	145	0	145	0	134	0
2	107	94750	119	155388	241	234198
3	136	254616	117	334391	253	312401
4	108	248344	117	825514	290	381349
5	106	835574	103	923171	307	407491

Table 5.7: Results for arbitrator_vs_frame.bp — 4096170 states. The protocol automatons creation takes 35 seconds.

The simplest approach for splitting the state space gives surprisingly the best results. The reassignment strategy is definitely the worst one. Recall that we are sending all edges that are crossing machine boundary through the server in all cases. We are doing this just because of support for reassignment. If we stoped this support it would bring improvements in other cases. At this point, I must say that the sending of some states through the server was not a bottle-neck during the presented tests. The reason is that we are using relativelly small number of clients. However, there is still overhead caused by sending of the states through the server which can be avoided by sending the states directly to the clients. The server becomes bottle neck when we use more clients — e.g. ten. This will be also removed by direct sending.

The memory utilization depends on the assignment of state space parts to the particular machines. If the number of reachable states assigned to the machines are comparable then we are able to use all memory. However, if the assignment is bad and an arbitrary machine uses all its memory, the checking is terminated without result.

To summarize the results, an significant speedup was achieved even without distribution. The distribution brings us other improvement in some cases, but not always.

Chapter 6

Related work

All model checkers suffers from the state space explosion and the task distribution can help. The distributed version of the explicit Murφ verifier was presented in [15]. In this work, random assignment of particular states to available machines is used. Although the communication criteria is utterly omitted good results are presented. It is caused mainly by the fact that computation of neighbour states in Murφ is very demanding. In this work, it is proved that the random assignment provides uniform distribution of workload if the number of states is much higher than the number of machines.

In [14], a distributed version of the SPIN model checker is described. Its main goal is to employ all memory of the available computers, not necesarily to have the result sooner. As the enumeration of state neighbours is quite fast in SPIN, the completely random assignment does not give good results. The authors propose another assignment. As it was explained in the previous chapter, SPIN state identifier is a vector. Each component identifies a state of one process. The assignment function uses only one component of the vector. The machine boundary is crossed only when the state of one particular process is changed. But there is typically many processes in the system and each transition means a change of only one or two of them. This way, lower communication requirements are achieved. This approach is similar to *static assignment* used in this work. The difference is that while the distributed SPIN checker creates 'slices' of the state space of width equal to one, we are using much wider ones. On the other side, a randomness of the slice assignment brings the uniform work distribution.

The problem with reporting error traces is solved by passing a list of events together with the transmitted state identifier. This list contains states that occur on the path from the initial state to the state, which contains the error.

Both works are not able to check properties that need data from more than one state. They can check assertions, reachability, deadlocks and invariants for which information contained in one state is enough. An example of such property, which is not observable from one state, is *infinite activity* or LTL formulas.

The parallel checking of LTL formulas is more complicated. The problem is that in a distributed environment the order of the visited nodes is not strictly given by the traversal strategy used. It depends on different speeds of the machines and communication delays. It is non-deterministic. In [19], the approach from [14] is improved in such way that information about strict DFS ordering is transferred together with the states. Another approach from [20] uses cycle detection based on the BFS traversal. The machines are synchronized when all states of the same depth are visited. The assignment of the states to the machines is still the same as in [14].

There are also works on paralelizing of the symbolic BDD based model checkers. In [21], partitioning of the BDD transition function and the BDD set is presented. The state space is divided into slices this way. The computation on a single slice of a set usually requires less memory than computation on the whole set. The set of newly achieved states is computed in parallel. The machines are synchronized at each level and exchange the states that were not generated by machine owning the slice it belongs to. At this moment, the memory requirements are compared and the state space can be resliced.

However, the BDD approach is not very suitable for model-checking of *behavior protocols*. The state of each component consists only from its program counter. It typically depends on program counters of other components. The BDD-based checking is not suitable for systems containing many dependent variables [22]. The model described by *behavior protocols* typically does not have other than dependent variables.

Chapter 7

Conclusion

A new approach for behavior protocols checking was presented. It better suits to distributed environment than the PTA-based checker. Moreover, the tests show that it behaves better even in the non-distributed case.

A prototype implementation used to compare different approaches for the state space division was created. This implementation achieves better results than the old PTA-based implementation of the checker when running on a single machine. The distribution brings other improvement in many cases, but not always.

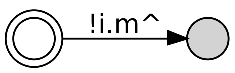
There are still many open issues related to this work. First the implementation should be changed to support only the promissing approaches. This should improve scalability.

Another big area for future work are state space division heuristics. An other important thing is extending this checker to be able to check more properties — namely the infinity activity or LTL formulas. The models accepted by the previous PTA-based checker also support synchronization. Some kind of synchronization, not necessarily the same atomic actions as these in the PTA-based checker, would be useful.

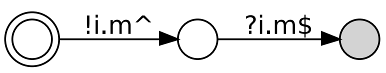
The checker implementation could be also improved in many ways. Minimization of protocol FSMs could decrease amount of required memory. The server process could be managed using Java JMX.

Appendix A

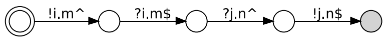
FSM representation of behavior protocols



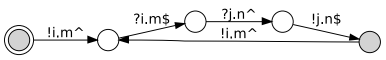
(a) Protocol `!i.m^` consisting of one event



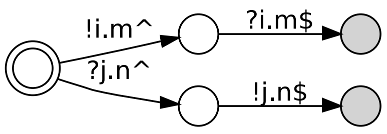
(b) Protocol `!i.m` using method call abbreviation.



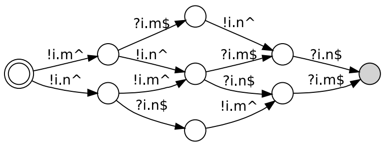
(c) Sequential operator - `!i.m;?j.n`



(d) Repetition operator - `(!i.m;?j.n)*`

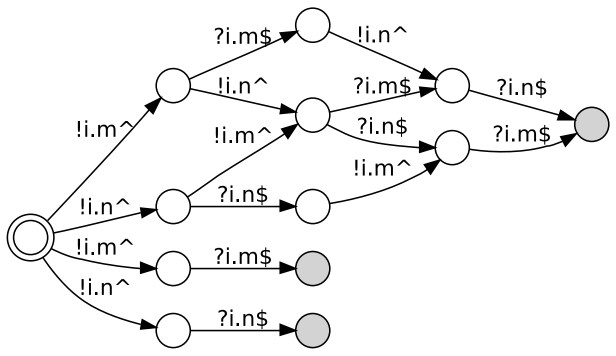


(e) Alternative operator - `!i.m+?j.n`

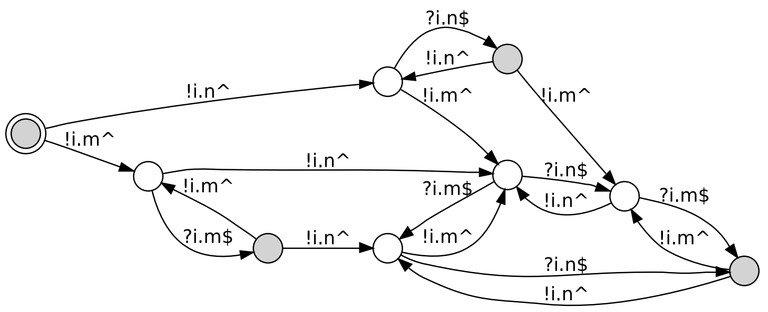


(f) And-parallel operator - `!i.m|!i.n`

Figure A.1: Simple behavior protocols



(a) Or-parallel operator `!i.m||!i.n`



(b) Protocol `(!i.m)*|(!i.n)*`

Figure A.2: Simple behavior protocols

Appendix B

Input file grammar

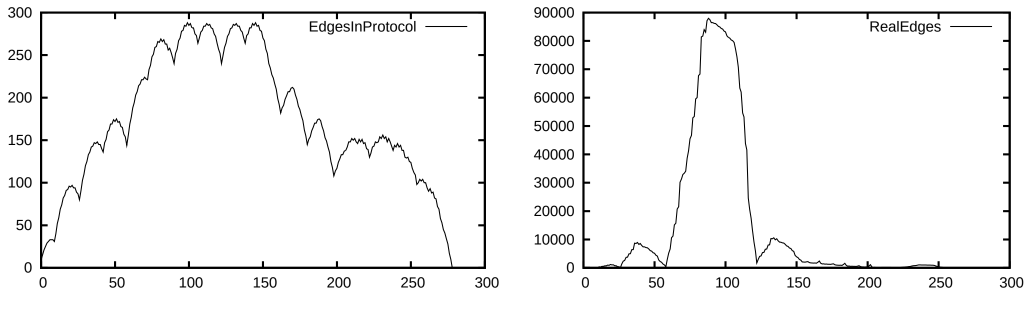
```
<ARCHITECTURE> ::= <PROTOCOL>
                |  <ARCHITECTURE> <BINDING> <PROTOCOL>
                |  <ARCHITECTURE> <BINDING>

<PROTOCOL> ::= <PROTOCOL_EXPR>
            |  "frame:" <PROTOCOL_EXPR>

<BINDING> ::= "sync{"<EVENT_LIST> "}"
           |  "sync{}"

<EVENT_LIST> ::= <EVENT>
              |  <BINDING> "," <EVENT>

<EVENT> ::= "?" <METHOD_NAME> "^"
         |  "!" <METHOD_NAME> "^"
         |  "?" <METHOD_NAME> "$"
         |  "!" <METHOD_NAME> "$"
         |  "?" <METHOD_NAME>
         |  "!" <METHOD_NAME>
         |  "?" <METHOD_NAME> "{" <PROTOCOL_EXPR> "}"
         |  "!" <METHOD_NAME> "{" <PROTOCOL_EXPR> "}"

<METHOD_NAME> ::= <IDENTIFIER> "." <IDENTIFIER>

<PROTOCOL_EXPR> ::= <PROTOCOL_EXPR> "+" <SEQUENCE>
                 |  <SEQUENCE>
```

```
<SEQUENCE> ::= <SEQUENCE> ";" <AND_PARALLEL>
            |  <AND_PARALLEL>

<AND_PARALLEL> ::= <AND_PARALLEL> "|"  <OR_PARALLEL>
                |  <OR_PARALLEL>

<OR_PARALLEL> ::= <OR_PARALLEL> "||" <REPETITION>
               |  <REPETITION>

<REPETITION> ::= <BRACKET> "*"
              |  <BRACKET>

<BRACKET> ::= "(" <PROTOCOL_EXPR> ")"
           |  <EVENT>
```

Appendix C

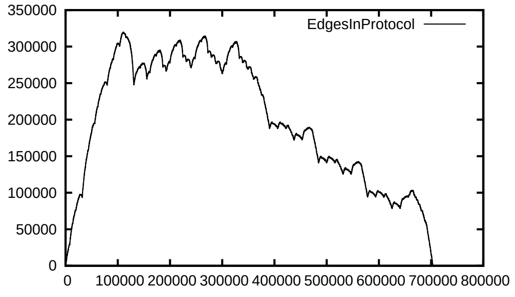
Edge and node density

Figures C.1 - C.4 show the number of edges coming from the one part of the the FSM to other. The left image is the graph of $E_P(l)$ while the right image always shows $CE_P(l)$. These functions are defined in Section 3.7. The heuristic proposed in that Section depends on the close relation between these functions. The following graphs show that there is no obvious relation. The data comes from `toplevel_frames.bp`. The other architectures from the case study look similar.

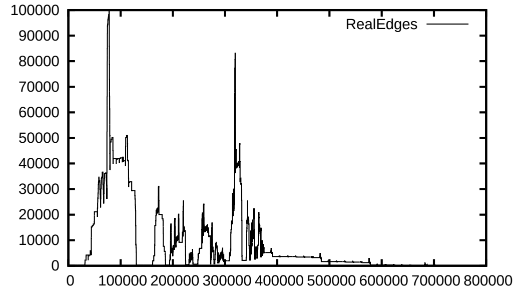


(a) (b)

Figure C.1: FSM of frame protocol

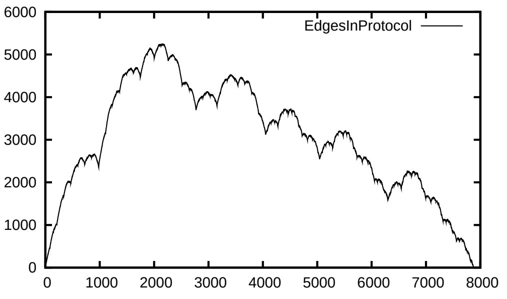


(a)

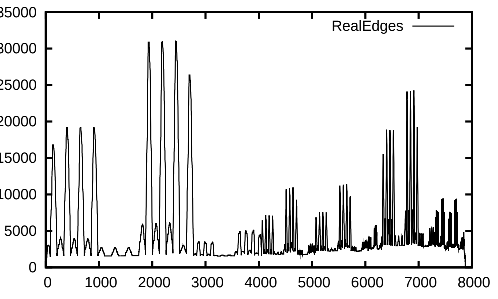


(b)

Figure C.2: Protocol FSM of Arbitrator component

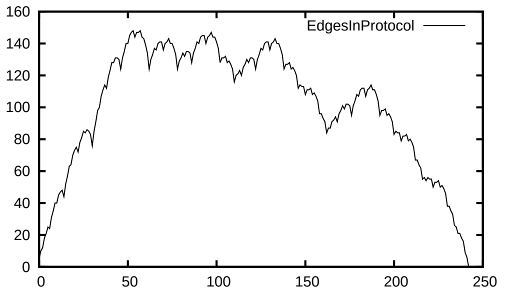


(a)

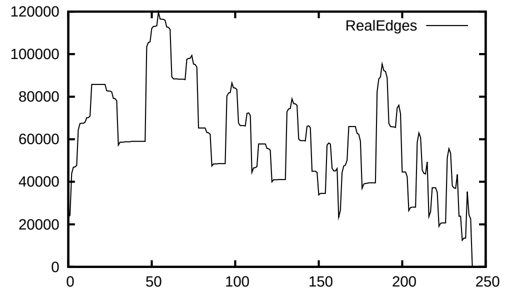


(b)

Figure C.3: Frame protocol of composite component containing Token, Card-Center and AccountDB components

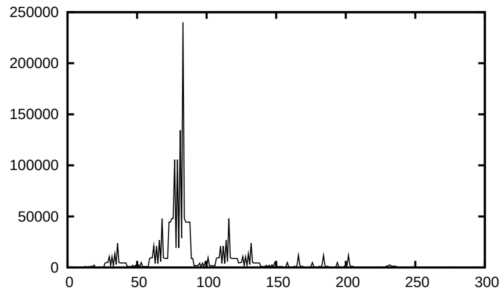


(a)

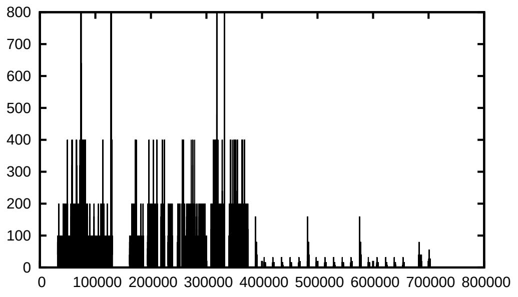


(b)

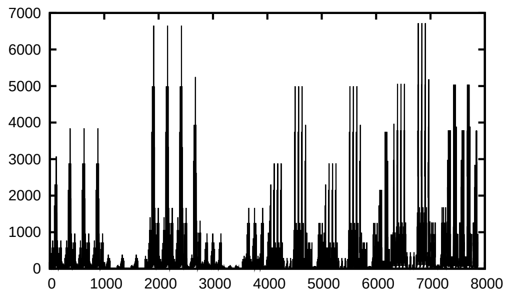
Figure C.4: Protocol FSM of Firewall component



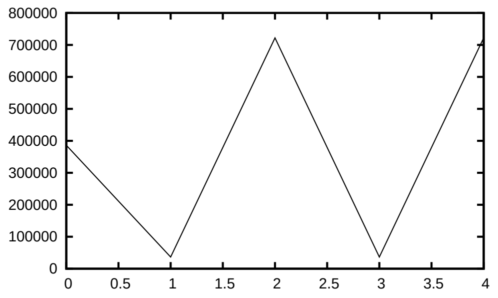
(a) Usage of frame protocol states



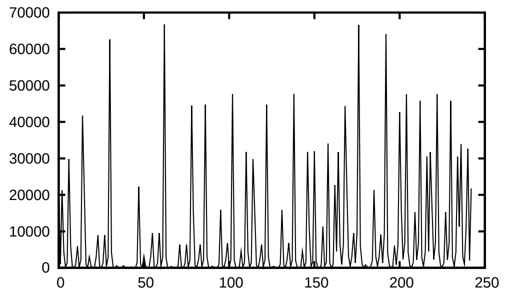
(b) Usage of arbitrator states



(c) Usage of composite component states



(d) Usage of FrequentFlyerDatabase states



(e) Usage of Firewall states

Figure C.5: These graphs capture how many times was a particular state of the protocol FSM involved in the state from the resultant FSM. The same architecture as in previous case was used. These graphs are useful to value particular division from the point of memory utilization. The integral over the parts assigned to different machines should be similar.

Bibliography

[1] F. Plasil and S. Visnovsky. Behavior Protocols for Software Components. *IEEE Transactions on Software Engineering*, 28(11), Nov 2002.

[2] M. Mach, F. Plasil, and J. Kofron. Behavior Protocol Verification: Fighting State Explosion. *International Journal of Computer and Information Science*, 6(1), Mar 2005.

[3] SOFA: Software Appliances - component framework `http://nenya.ms.mff.cuni.cz`.

[4] SPIN model checker - `http://www.spinroot.com`.

[5] I. Beer, S. Ben-David, C. Eisner, and A. Landver. Rulebase: An industry-oriented formal verification tool. In *33rd Design Automation Conference*, page 655–660, 1996.

[6] Java PathFinder - `http://javapathfinder.sourceforge.net`.

[7] Shoham Ben-David, Cindy Eisner, Daniel Geist, and Yaron Wolfsthal. Model checking at IBM. *Verification and Testing Technologies in System Design*, 22(2), 2003.

[8] R. Sethi A. V. Aho and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988.

[9] J. Kofron. Enhancing Behavior Protocols with Atomic Actions. Technical Report 2006/2, Dep. of SW Engineering, Charles University, Jan 2006.

[10] O. Sery. Model Checking and Reduction of Behavior Protocols, Master Thesis, Charles University in Prague, May 2006.

[11] P. Parizek, F. Plasil, and J. Kofron. Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model

Checker. In *Proceedings of 30th IEEE/NASA Software Engineering Workshop (SEW-30)*, Apr 2006.

[12] J. Adamek and F. Plasil. Behavior Protocols: Tolerating Faulty Architectures and Supporting Dynamic Updates. Technical Report 2002/10, Department of Computer Science, University of New Hampshire, Oct 2006.

[13] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[14] Flavio Lerda and Riccardo Sisto. Distributed-memory model checking with SPIN. In *Proc. of the 5th International SPIN Workshop*, volume 1680 of *LNCS*. Springer-Verlag, 1999.

[15] Ulrich Stern and David L. Dill. Parallelizing the Murφ Verifier. In *Computer Aided Verification*, pages 256–278, 1997.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns,Elements of Reusable Object-Oriented Software.*

[17] P. Jezek, J. Kofron, and F. Plasil. Model Checking of Component Behavior Specification: A Real Life Experience. In *Proceedings of International Workshop on Formal Aspects of Component Software (FACS'05), Macao*, Oct 2005.

[18] J. Adamek, T. Bures, P. Jezek, J. Kofron, V. Mencl, P. Parizek, and F. Plasil. Real-life Behavior Specification of Software Components, Presented at the 11th EMEA Academic Forum, Dublin, Ireland, May 2006.

[19] J. Barnat, L. Brim, and J. Stříbrná. Distributed LTL Model-Checking in SPIN. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, volume 2057 of *LNCS*, pages 200–216. Springer, 2001.

[20] J. Barnat, L. Brim, and J. Chaloupka. Parallel Breadth-First Search LTL Model-Checking. In *18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 106–115. IEEE Computer Society, Oct. 2003.

[21] T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In *Proc. of the 12th International Conference on Computer Aided Verification. Springer-Verlag*, June 2000.

[22] Alan J. Hu and David L. Dill. Reducing BDD size by exploiting functional dependencies. In *Design Automation Conference*, pages 266–271, 1993.