1

Model Checking with Abstraction for Web Services

Abstract

Web-services are highly distributed programs, and thus, prone to concurrency-related errors. Model checking is a powerful technique to identify flaws in concurrent systems. However, the existing model checkers have only very limited support for the programming languages and communication mechanisms used by typical implementations of web services. This chapter presents a formalization of communication semantics geared for web services, and an automated way to extract formal models from programs implementing web services for automatic formal analysis. The formal models are analyzed by means of a symbolic model checker that implements automatic abstraction refinement. Our implementation takes one or more PHP5 programs as input, and is able to verify joint properties of these programs running concurrently.

1.1 Introduction

Web-services are instantiations of service-oriented architectures, where a service is a function that is well-defined, self-contained, and does not depend on the context or state of other services [1]. They are designed to be published, accessed and used via intra- or Internet. The elements of the design are 1) a service provider, which offers some service; 2) a service broker who maintains a catalog of available services; and 3) service requesters which seek for a service from the service broker, and then attach to the service provider by composing the offered services with its own components. A web service offers an arbitrary complex functionality, which is described in a global system structure. Examples of web services include information systems such as map or travel services, e-commerce systems such as web shops, travel agencies, stock brokers, etc. Clearly, it is essential to enforce security and safety requirements in the development of such systems.

Web-services are typically implemented in a very distributed manner, and thus, are prone to errors caused by the distribution. They often involve mul-

tiple parties. As an example, consider an online shop that accepts charges to a credit card as form of payment. The parties involved are the users or customers, the vendor itself, and back-office service providers, e.g., the payment clearing service. The authorization of the payment is given by a service company for credit card transactions, whereas the "shopping basket" and warehousing are implemented by the vendor. It is easy to imagine another party involved in a transaction, e.g., a company that performs the actual shipment.

Each party typically employs a large set of machines for the purpose of load sharing. The safety and security requirements are often global, i.e., require reasoning about multiple parties and may involve more than one server at each party.

A typical scenario is depicted in Fig. 1.1: a merchant is operating three hosts (e.g., for redundancy or load-balancing reasons). Two of these hosts ('Host 1' and 'Host 2') are used to run a web server, e.g., Apache. The web server itself is split up into multiple processes $T_1, \ldots, T_4$. The web server processes have joint access to a shared database, e.g., using the SQL protocol. This database is assumed to be the only form of communication between the server processes. The server processes may contact a third party, e.g., to authorize a payment. Incoming client requests are modeled by means of processes T_5 and T_6.

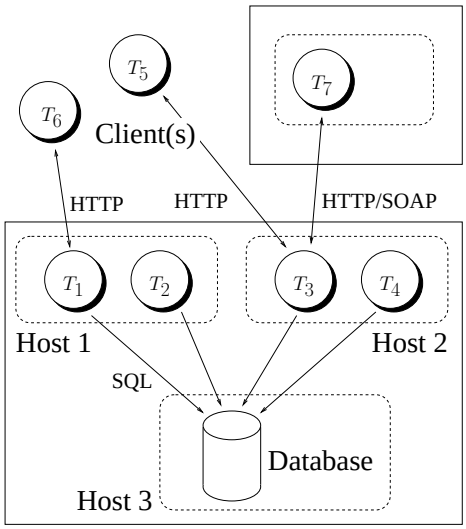

Fig. 1.1. A typical scenario: A merchant operating multiple machines offering a service to multiple clients, and communicating with a third party.

Analyzing software that implements such services therefore requires reasoning about many programs running in parallel. Concurrent software is noto-

riously error-prone. Approaches based on testing often fail to find important concurrency bugs.

Model checking [2, 3] is a formal verification technique. It has been shown to be especially useful for verifying concurrency-related properties, and identifying bugs related to process schedules. In the context of web services, there are manifold properties that model checking can be used to verify:

- Safety properties, e.g., that no exceptions are thrown by the code, that the code is free of data races, or that certain security properties hold,
- Liveness properties, e.g., that the code does not get into a state in which it deadlocks or livelocks.

However, model checking suffers from the *state explosion problem*. In case of BDD-based symbolic model checking, this problem manifests itself in the form of unmanageably large BDDs [4]. In case of concurrent software, the state-space explosion problem comes from two sources: 1) The model checker has to consider manifold interleavings between the threads, and 2) software usually operates on a very large set of data variables.

The principal technique to address the first problem is partial order reduction [5]. The principal method for addressing the large amount of data is *abstraction*. Abstraction techniques reduce the state space by mapping the set of states of the actual, concrete system to an abstract, and smaller, set of states in a way that preserves the relevant behaviors of the system. The use of abstraction on transition systems is formalized by the abstract interpretation framework [6].

Predicate abstraction [7, 8] is one of the most popular and widely applied methods for systematic abstraction of programs. It abstracts data by only keeping track of certain predicates on the data. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. Verification of a software system with predicate abstraction consists of constructing and evaluating a finite-state system that is an abstraction of the original system with respect to a set of predicates.

Typically, this abstract program is created using *Existential Abstraction* [9]. This method defines the instructions in the abstract program so that it is guaranteed to be a *conservative* over-approximation of the original program for reachability properties. Thus, in order to show that no erroneous state is reachable, it is sufficient to show that the abstract model does not contain it.

The drawback of such a conservative abstraction is that when model checking of the abstract program fails, it may produce a counterexample that does not correspond to any counterexample on the original program. This is usually called a *spurious counterexample*. When a spurious counterexample is encountered, *refinement* is performed by adjusting the set of predicates in a way that eliminates this counterexample from the abstract program. This process is iterated until either a counterexample is found, or the property is shown. The

actual steps of the loop follow the *counterexample guided abstraction refinement* (CEGAR) framework.

The CEGAR framework has been implemented in most software model checking tools (e.g., SLAM [10, 11], MAGIC [12], BLAST [13], SATABS [14]). The existing software Model Checkers, however, are not readily applicable for most programs implementing web services. This is due to the fact that the existing tools lack of support for the programming languages and the communication primitives used for web services.

A number of programming languages has been designed specifically for implementing web services. The commonly used programming languages for web-applications are WSDL, BPEL, PHP [15], and ASP [16]. While in general their goal is to provide the programming constructs for the implementation of web services, they differ in the level at which they address the web service operations. For example, BPEL [17] has been developed to specify interaction protocols (synchronous and asynchronous) between web services. BPEL is also a high-level language for implementing web service applications, and is supported by the most industrial players in the field (IBM, Oracle, BEA). It is designed for specifying the communication among service participants and its users. Its disadvantage is that it does not support the low level implementation details of the service functionality.

Among the programming languages that support low-level details of web service implementations are ASP and PHP. ASP (Active Server Pages) is based on either Visual Basic Script or JScript. ASP is a proprietary system that is natively implemented only on Microsoft Internet Information Server (IIS). There are attempts of implementations of ASP on other architectures, e.g., InstantASP from Halcyon and Chili!Soft ASP. Formal models of ASP scripts are difficult to generate as ASP permits full access to the WIN32 API, which offers an enormous amount of functions.

The other commonly used programming language for web-applications is PHP, a scripting language specialized for the generation of HTML, server-side JAVA, Microsoft's ASPx, and more recently, C# as part of .NET. PHP is an interpreted programming language that has a C-like syntax. The most commonly used platform for PHP is the Apache web server, but there are implementations for IIS-based servers as well.

A large number of tools and techniques for modeling and model checking BPEL processes have been developed (see for example, [18, 19, 20, 21, 22, 23, 24, 25]). They focus on analyzing the interaction protocols, the orchestration, and the composition of web services. They are not applicable, however, to verifying the actual applications that implement the web services due to the restrictions of the BPEL notation. As far as the verification of the actual implementations of web services, there are no tools available yet. Moreover, to the best of our knowledge there are no implementations of the abstraction-refinement framework available for any of the languages that are typically used for implementations of web services.

While there are model checkers for concurrent Java, the concurrency is assumed to be implemented using the Java thread interface. Communication between the processes is assumed to be implemented by means of shared data. In contrast to that, programs implementing web services are usually *single threaded*. The concurrency arises from the fact that there are multiple instances of the same single-threaded program running. Communication between the processes is typically implemented by either

1. TCP sockets using protocols such as HTML or XML,
2. shared databases using query languages such as SQL.

Note that the two communication primitives above have different semantics: in the context of web services, communication through a TCP socket is usually done in a *synchronous*, blocking way; after sending data, the sending process usually waits for an acknowledgment by the receiver and thus, is blocked until the receiving process accepts and processes the message.

In contrast to that, database accesses are usually performed independently by each process. Blocking is avoided in order to obtain scalability in the number of database clients. While the SQL protocol itself is blocking, the communication between processes through a shared database has *asynchronous* semantics. Thus, the interleavings of competing accesses to the database become relevant.

Returning to the scenario described above (Fig. 1.1), shared databases are usually only accessible within the realm of a single party. Links to external parties (clients, third-party service providers) are typically implemented using synchronizing protocols such as SOAP.

Formal reasoning about global properties of web services requires identification and modeling of *both* of these communication primitives. In particular, support is needed for both synchronizing and asynchronous inter-process communication. None of the existing software model checkers provides such support.

We propose to use *Labeled Kripke Structures* (LKS) as means of modeling web services: LKSs are directed graphs in which states are labeled with atomic propositions and transitions are labeled with actions. The synchronization semantics is derived from CSP (Communicating Sequential Processes), i.e., processes synchronize on shared events and proceed independently on local ones. The formalism supports shared variables. Once the formal model is extracted from the implementation, the web service becomes amenable to formal analysis by means of model checking [26].

Contribution

This article addresses a problem of verifying the applications that implement the web services and develops techniques for modeling and verification of low-level languages used for the implementation of web services.

We formalize the semantics of a PHP-like programming language for web services by means of labeled Kripke structures. We use a computational model

that allows both synchronizing and interleaving communication. Previous models are limited to either synchronous or asynchronous inter-process communication. Once the model is obtained, automatic predicate abstraction is applied to formally analyze the web services. Manual, and thus, error-prone, generation of models is no longer needed.

We implement the technique described above in a tool called SatAbs. It is able to check safety properties of a combination of multiple PHP scripts. It uses the Zend 2 engine as a front-end for PHP. The abstract model is computed using SAT, and analyzed using a symbolic model checker that features partial order reduction.

Related Work

Formal models for synchronization mechanisms have been thoroughly explored. For example, CSP [27] and the calculus of communicating systems (CCS for short) [28] were introduced in the same years and influenced one another throughout their development. CSP and CCS allow the description of systems in terms of component processes that operate independently, and interact with each other solely through different synchronization mechanisms. In CSP, two processes must synchronize on any identically named action (i.e., by means of shared actions) that both are potentially capable of performing. Moreover, any number of processes may interact on a single shared action. In CCS, processes may interact on two complementary actions (e.g., a and $\overline{a}$, respectively input and output action over a shared channel named a), and only bi-party interaction is supported.

Both CSP and CCS do not provide a way to directly represent and reason about dynamic communications topologies and migrating computational agents, which are an important aspect of many modern systems. Some people see this as a major drawback of their theory. The pi-calculus [29] arose as a generalization of CCS [28]. In the pi-calculus, not only processes synchronize on two input/output actions over a shared channel, they are also able to send data along those channels.

This paper builds on work described in [26], where SAT-based predicate abstraction is applied to a SystemC design. SystemC is a description language based on C++ that is used to model both hardware and software components of embedded designs. The concurrency primitives of SystemC are modeled using the state/event-based notation introduced in [30]. As in this work, the modeling framework consists of labeled Kripke structures (LKSs).

The combined state-based and event-based notation has been explored by a number of researchers. De Nicola and Vaandrager [31], for instance, introduce 'doubly labeled transition systems', which are very similar to our LKSs. Kindler and Vesper [32] use a state/event-based temporal logic for Petri nets. Abstraction-based model checking is not reported for these formalizations. Huth *et al.* [33] also propose a state/event framework, and define rich notions of abstraction and refinement. In addition, they provide 'may' and 'must'

modalities for transitions, and show how to perform efficient three-valued verification on such structures.

Most of the related work on formal analysis of web services consists of verifying of a formal description of the web service using a model checker. For example, in [34] the authors propose translating models described in BPEL into Promela and check the web service flow with the SPIN model-checker. Another example of modeling and model checking of BPEL protocols is [18]. It uses Petri-nets for modeling and verification of coordination of the BPEL processes.

In [35], a similar approach uses NuSMV. The verification of Linear Temporal First-Order properties of asynchronously communicating web services is studied in [36]. The peers receive input from their users and asynchronous messages from other peers. The authors developed a special purpose model checker [36] that allows verification of Web applications specified in WebML.

Among other major techniques for analyzing web services there are works of Bultan and others. This group developed a formal model for interactions of composite web services supported by techniques for analysis of such interactions [25, 24, 23, 37, 22]. Kramer et al. [38] define a model-based approach to verifying process interactions for coordinated web service compositions. The approach uses finite state machine representations of web service orchestrations and assigns semantics to the distributed process interactions. Pistore et al. proposed techniques for the automated synthesis of composite web services from abstract BPEL components [20], and verification of Web service compositions defined by sets of BPEL processes [19]. The modeling techniques are adopted for representing the communications among the services participating in the composition. Indeed, these communications are asynchronous and buffered in the existing execution frameworks, while most verification approaches assume a synchronous communication model for efficiency reasons.

In [39] at least the interface specification is verified at the source code level using Java PathFinder. The SPIN model-checker is used for the behavior verification of the asynchronously communicating peers (bounded message queues). A language for specifying web service interfaces is presented in [40]. None of the above techniques uses automated abstraction-based verification and thus, are less competitive in verification of large-scale web systems.

Outline

We provide background information on PHP and related languages and predicate abstraction in section 1.2. We explain the computational model in section 1.3 and formalize the semantics of the subset of PHP we handle. Section 1.4 provides details on how to abstract the generated model and how to verify it.

1.2 Background

1.2.1 Implementations of Web-Services

A web service is a system that provides an interface defined in terms of XML messages and that can be accessed over the Internet [41]. It is intended for machine-to-machine interaction.

Such a service is usually embedded in an application server. The application server is a program that runs in an infinite loop and waits until a client connects via a socket (by means of bidirectional communication over the Internet) to a specified port. In order to be able to process several requests simultaneously, each incoming request is handled by a new thread from a thread pool. Thus, there might be multiple instances of the same code running concurrently.

There are three main XML-based standards that define the web services architecture: the Universal Description, Discovery and Integration (UDDI) protocol is used to publish and discover web services, which are specified in the Web Service Description Language (WSDL). The communication between web services is defined by the Simple Object Access Protocol (SOAP).

There are three major roles within the web service architecture (Fig. 1.2):

Service provider The service provider implements the service and makes it available on the Internet.

Service requester The client that connects to the web service.

Service broker This is a logically centralized directory of services where service providers can publish their services together with a service description. The service requester can search for services in this directory.

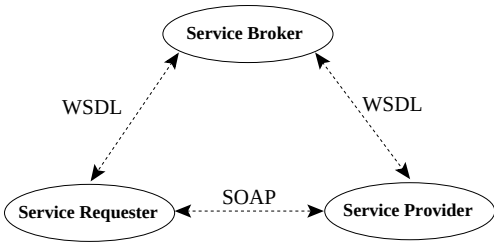

Fig. 1.2. Web service actors and protocols

We restrict the presentation to service providers and requesters, i.e., we assume that services are addressed statically by the requesters. Since there are XML tools for nearly every operating system and every programming language, web services are independent from the machine architecture, the operating system and the programming language. We use PHP syntax to present our formalism. The formalism is applicable to similar languages for

web-services as well with minor modifications specific to the syntax of those languages.

1.2.2 Synchronous Communication

Synchronous communication within web services is characterized by the client being blocked until the service request has been processed.

Example 1.1. An example of synchronous communication is a credit card service used in an e-commerce application: when the customer checks out his shopping cart, the credit card service is invoked and the application then *waits* for the approval or denial of the credit card transaction (see Fig. 1.3). Figure 1.4 and Figure 1.5 show an example of how a client and a server might be implemented, respectively.

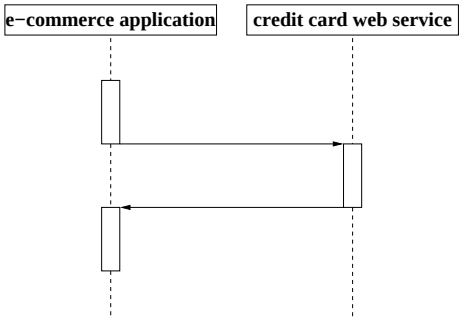

Fig. 1.3. Synchronous communication

```
$client = new
  SoapClient("http://example.net/soap/urn:creditcard.wsdl");
if($client->debit($cardnumber, $amount) ) {
  // transaction approved
  // ...
} else {
  // transaction failed
  // ...
}
```

Fig. 1.4. Example of SOAP client code

```
class CreditCardTransaction {
  function debit($cardnumber, $amount) {
    // debit money from credit card and return error code
    // ...
    return $success;
  }
}

$server = new SoapServer("creditcard.wsdl");
$server->setClass("CreditCardTransaction");
$server->handle();
```

Fig. 1.5. Example of SOAP server code

1.2.3 Asynchronous Communication

Asynchronous communication is used when the program cannot wait for the receiver to acknowledge that the data was received. As an example, consider the part of the e-commerce application that maintains a shopping basket. This shopping basked is typically stored in persistent memory, e.g., a database. The information is typically accessed by multiple processes over the lifetime of the interaction with the client, and in a sense, communicated from one instance of the server process to the next.

The time that passes between the accesses to the basket are arbitrary. Synchronization between the time the data is sent (stored) and received (read) is not desired. Also note that the order of operations becomes important: as an example, assume that a customer simultaneously issues a request to add an item of a particular type to the basket, and independently, another request to remove all items of that same type. The final result (none or one item of that type) depends on the order in which the database transactions are processed.[1]

We assume that a shared database is the only means to exchange data among the processes in a non-synchronizing way, i.e., we do not model local, shared files that can be written into, or the like. The database is expected to guarantee a specific level of atomicity in the form of transactions. The different transactions from the various processes accessing the database are assumed to interleave arbitrarily. The issue of ordering is captured by the concept of *races*. The shopping-basked described above is an instance of such a race. Such races often represent erroneous behavior. Bugs caused by race conditions are a well-known problem of any system with asynchronous concurrency and are very difficult to detect by means of testing.

[1] Synchronous communication, as described above, may in principle be implemented by means of a database. However, the resulting implementation would need to rely on polling, and thus, is very unlikely to scale.

1.2.4 Predicate Abstraction

The abstraction refinement process using predicate abstraction has been promoted by the success of the SLAM project at Microsoft Research, which aims at verifying partial correctness of Windows device drivers [10]. The algorithm starts with a coarse abstraction, and if it is found that an error-trace reported by the model checker is not realistic, the error trace is used to refine the abstract program, and the process proceeds until no spurious error traces can be found. The actual steps of the loop follow the *counterexample guided abstraction refinement* (CEGAR) paradigm and depend on the abstraction and refinement techniques used. Assume that a program M consists of components $M_1, \ldots, M_n$ executing concurrently. The verification procedure checks if a property φ holds for M by using the following three-step iterative process:

1. **Abstract.** Create an abstraction $\widehat{M}$ such that if M has a bug, then so does $\widehat{M}$. This can be done component-wise without constructing the full state space of M.
2. **Verify.** Check if a property φ holds for $\widehat{M}$. If yes, report success and exit. Otherwise let $\widehat{C}$ be a counterexample that indicates where φ fails in $\widehat{M}$.
3. **Refine.** Check if $\widehat{C}$ is a valid counterexample with respect to M. This step is called *simulation*. Again, this can be done component-wise. If $\widehat{C}$ corresponds to a real behavior then the algorithm reports the flaw and a fragment of each M_i that shows why the property is not satisfied ($M \not\models \varphi$). Otherwise, $\widehat{C}$ is spurious, and $\widehat{M}$ is refined using $\widehat{C}$ to obtain a more precise abstraction. The algorithm continues with step 1.

Existential Abstraction

The goal of *abstraction* is to compute an abstract model $\hat{M}$ from the concrete model M such that the size of the state-space is reduced and the property of interest is preserved. We denote the set of abstract states by $\hat{S}$. A concrete state is mapped to an abstract state by means of an *abstraction function*, which we denote by $\alpha : S \longrightarrow \hat{S}$. We also extend the definition of α to sets of states: Given a set $S' \subseteq S$, we denote $\{\hat{s} \in \hat{S} \mid \exists s \in S'.\alpha(s) = \hat{s}\}$ by $\alpha(S')$.

We restrict the presentation to *reachability properties*. The goal therefore is to compute an abstraction that preserves reachability: any program location that is reachable in M must be reachable in $\hat{M}$. *Existential Abstraction* is a form of abstraction that preserves reachability [9].

Definition 1.2 (Existential Abstraction [9]).
Given an abstraction function $\alpha : S \longrightarrow \hat{S}$, a model $\hat{M} = (\hat{S}, \hat{S}_0, \hat{R})$ is called an Existential Abstraction *of $M = (S, S_0, R)$ (here $\hat{S}_0, S_0, \hat{R}, R$ are the initial states and transitions functions of $\hat{M}$ and M respectively) iff the following conditions hold:*

1. *The abstract model can make a transition from an abstract state $\hat{s}$ to $\hat{s}'$ iff there is a transition from s to s' in the concrete model and s is abstracted to $\hat{s}$ and s' is abstracted to $\hat{s}'$:*

$$\begin{aligned} &\forall \hat{s}, \hat{s}' \in (\hat{S} \times \hat{S}).(\hat{R}(\hat{s}, \hat{s}') \iff \\ &\quad (\exists s, s' \in (S \times S).R(s, s') \wedge \\ &\quad \alpha(s) = \hat{s} \wedge \alpha(s') = \hat{s}')) \end{aligned} \tag{1.1}$$

2. *An abstract state $\hat{s} \in \hat{S}$ is an initial state iff there exists an initial state s of M that is abstracted to $\hat{s}$:*

$$\forall \hat{s} \in \hat{S}.(\hat{s} \in \hat{S}_0 \iff \exists s \in S_0.\alpha(s) = \hat{s}) \tag{1.2}$$

Existential abstraction is a conservative abstraction with respect to reachability properties, which is formalized as follows:

Theorem 1.3. *Let $\hat{M}$ denote an existential abstraction of M, and let ϕ denote a reachability property. If ϕ holds on $\hat{M}$, it also holds on M:*

$$\hat{M} \models \phi \Longrightarrow M \models \phi$$

Thus, for an existential abstraction $\hat{M}$ and any program location l that is not reachable in the abstract model $\hat{M}$, we may safely conclude that it is also unreachable in the concrete model M. Note that the converse does not hold, i.e., there may be locations that are reachable in $\hat{M}$ but not in M.

Notation

We denote the set of program locations by L. In program verification, the abstract transition relation $\hat{R}$ is typically represented using a partitioning, similarly to the concrete transition relation R. The abstract transition relation for program location $l \in L$ is denoted by $\hat{R}_l(\hat{s}, \hat{s}')$, and the program location of an abstract state $\hat{s}$ by $\hat{s}.\ell$.

$$\hat{R}(\hat{s}, \hat{s}') \iff \bigwedge_{l \in L} (s.\ell = l \longrightarrow \hat{R}_l(\hat{s}, \hat{s}')) \tag{1.3}$$

The computation of $\hat{R}$ follows the structure of the partitioning according to the program locations, i.e., $\hat{R}$ is generated by computing $\hat{R}_l$ from R_l for each location $l \in L$ separately. The following sections describe algorithms for computing $\hat{R}_l$.

Predicate Abstraction

There are various possible choices for an abstraction function α. *Predicate Abstraction* is one possible choice. It is one of the most popular and widely applied methods for systematic abstraction of programs, and was introduced by Graf and Saïdi [7]. An automated procedure to generate predicate abstractions was introduced by Colón and Uribe [8]. Predicate abstraction abstracts data by only keeping track of certain predicates on the data. The predicates are typically defined by Boolean expressions over the concrete program variables. Each predicate is then represented by a Boolean variable in the abstract

program, while the original data variables are eliminated. Verification of a software system by means of predicate abstraction entails the construction and evaluation of a system that is an abstraction of the original system with respect to a set of predicates.

We denote the set of Boolean values by $\mathbb{B} := \{\mathsf{T}, \mathsf{F}\}$. Let $P := \{\pi_1, \ldots, \pi_n\}$ denote the set of predicates. An abstract state $\hat{s} \in \hat{S}$ consists of the program location and a valuation of the predicates, i.e., $\hat{S} = L \times \mathbb{B}^n$. We denote the vector of predicates by $\hat{s}.\pi$. We denote the value of predicate i by $\hat{s}.\pi_i$. The abstraction function $\alpha(s)$ maps a concrete state $s \in S$ to an abstract state $\hat{s} \in \hat{S}$:

$$\alpha(s) := \langle s.\ell, \pi_1(s), \ldots, \pi_n(s) \rangle \tag{1.4}$$

Example 1.4. As an example, consider the following program statement, where `i` denotes an integer variable:

```
i++;
```

This statement translates to the following concrete transition relation $R_l(s, s')$:

$$R_l(s, s') \iff s'.i = s.i + 1$$

Assume that the set of predicates consists of $\pi_1 \iff i = 0$ and $\pi_2 \iff \mathit{even}(i)$, where $\mathit{even}(i)$ holds iff i is an even number. With $n = 2$ predicates, there are $(2^n)^2 = 16$ potential abstract transitions. A naïve way of computing $\hat{R}$ is to enumerate the pairs $\hat{s}, \hat{s}'$ and to check Eq. 1.1 for each pair separately. As an example, the transition from $\hat{s} = (l, \mathsf{F}, \mathsf{F})$ to $\hat{s}' = (l, \mathsf{F}, \mathsf{F})$ corresponds to the following formula over concrete states:

$$\begin{aligned} \exists s, s'.\ & \neg s.i = 0 \wedge \neg \mathit{even}(s.i) \quad \wedge \\ & s'.i = s.i + 1 \quad \wedge \\ & \neg s'.i = 0 \wedge \neg \mathit{even}(s'.i) \end{aligned} \tag{1.5}$$

This formula can be checked by means of a decision procedure. For instance, an automatic theorem prover such as Simplify [42] can be used if a definition of $\mathit{even}(i)$ is provided together with Eq. 1.5. Since Eq. 1.5 does not have any solution, this abstract transition is not in $\hat{R}_l$. Fig. 1.6 shows the abstract transitions for the program statement above, and one corresponding concrete transition (i.e., a satisfying assignment to Eq. 1.1) for each possible abstract transition.

1.3 Extracting Formal Models of Web-Services

1.3.1 Computational Model

A labeled Kripke structure [30] (LKS for short) is a 7-tuple $(S, \mathit{Init}, P, \mathcal{L}, T, \Sigma, \mathcal{E})$ with S a finite set of *states*, $\mathit{Init} \subseteq S$ a set of initial states, P a finite set of

Abstract Transition				Concrete Transition	
$\hat{s}.\pi_1$	$\hat{s}.\pi_2$	$\hat{s}'.\pi_1$	$\hat{s}'.\pi_2$	s	s'
F	F	F	T	$s.i = 1$	$s'.i = 2$
F	F	T	T	$s.i = -1$	$s'.i = 0$
F	T	F	F	$s.i = 2$	$s'.i = 3$
T	T	F	F	$s.i = 0$	$s'.i = 1$

Fig. 1.6. Example for existential abstraction: Let the concrete transition relation $R_l(s, s')$ be $s'.i = s.i + 1$ and let $\pi_1 \iff i = 0$ and $\pi_2 \iff even(i)$. The table lists the transitions in $\hat{R}_l$ and an example for a corresponding concrete transition.

atomic state propositions, $\mathcal{L} : S \to 2^P$ a *state-labeling function*, $T \subseteq S \times S$ a transition relation, Σ a finite set (*alphabet*) of *events* (or *actions*), and $\mathcal{E} : T \to (2^\Sigma \setminus \{\emptyset\})$ a *transition-labeling function*. We write $s \xrightarrow{A} s'$ to mean that $(s, s') \in T$ and $A \subseteq \mathcal{E}(s, s')$.[2] In case A is a singleton set $\{a\}$ we write $s \xrightarrow{a} s'$ rather than $s \xrightarrow{\{a\}} s'$. Note that both states and transitions are 'labeled', the former with sets of atomic propositions, and the latter with non-empty sets of actions.

A *path* $\pi = \langle s_1, a_1, s_2, a_2, \ldots \rangle$ of an LKS is an alternating infinite sequence of states and actions subject to the following: for each $i \geqslant 1$, $s_i \in S$, $a_i \in \Sigma$, and $s_i \xrightarrow{a_i} s_{i+1}$.

The *language* of an LKS M, denoted $L(M)$, consists of the set of maximal paths of M whose first state lies in the set *Init* of initial states of M.

1.3.2 Abstraction

Let $M = (S, Init, P, \mathcal{L}, T, \Sigma, \mathcal{E})$ and $\hat{M} = (S_{\hat{M}}, Init_{\hat{M}}, P_{\hat{M}}, \mathcal{L}_{\hat{M}}, T_{\hat{M}}, \Sigma_{\hat{M}}, \mathcal{E}_{\hat{M}})$ be two LKSs. We say that $\hat{M}$ is an *abstraction* of M, written $M \sqsubseteq \hat{M}$, iff

1. $P_{\hat{M}} \subseteq P$,
2. $\Sigma_{\hat{M}} = \Sigma$, and
3. For every path $\pi = \langle s_1, a_1, \ldots \rangle \in L(M)$ there exists a path $\pi' = \langle s'_1, a'_1, \ldots \rangle \in L(\hat{M})$ such that, for each $i \geqslant 1$, $a'_i = a_i$ and $\mathcal{L}_{\hat{M}}(s'_i) = \mathcal{L}(s_i) \cap P_{\hat{M}}$.

In other words, $\hat{M}$ is an abstraction of M if the 'propositional' language accepted by $\hat{M}$ contains the 'propositional' language of M, when restricted to the atomic propositions of $\hat{M}$. This is similar to the well-known notion of 'existential abstraction' for Kripke structures in which certain variables are hidden [43].

Two-way abstraction defines an equivalence relation $\sim$ on LKSs: $M \sim M'$ iff $M \sqsubseteq M'$ and $M' \sqsubseteq M$. We shall only be interested in LKSs up to $\sim$-equivalence.

[2] In keeping with standard mathematical practice, we write $\mathcal{E}(s, s')$ rather than the more cumbersome $\mathcal{E}((s, s'))$.

1.3.3 Parallel Composition

Many properties of web-services can only be verified in the context of multiple processes. We expect that large amounts of data have to be passed between those processes. We therefore modify the notion of parallel composition in [30] to allow communication through shared variables. The shared variables are used to model both communication through a database and to model the data that is passed over sockets.

Let $M_1 = (S_1, \mathit{Init}_1, P_1, \mathcal{L}_1, T_1, \Sigma_1, \mathcal{E}_1)$ and $M_2 = (S_2, \mathit{Init}_2, P_2, \mathcal{L}_2, T_2, \Sigma_2, \mathcal{E}_2)$ be two LKSs. We assume M_1 and M_2 share the same state space, i.e., $S = S_1 = S_2$, $P = P_1 = P_2$, and $\mathcal{L} = \mathcal{L}_1 = \mathcal{L}_2$. We denote by $s \xrightarrow{A}_i s'$ the fact that M_i can make a transition from s to s'.

The parallel composition of M_1 and M_2 is given by $M_1 \parallel M_2 = (S, \mathit{Init}_1 \cap \mathit{Init}_2, P, \mathcal{L}, T, \Sigma_1 \cup \Sigma_2, \mathcal{E})$, where T and $\mathcal{E}$ are such that $s \xrightarrow{A} s'$ iff $A \neq \emptyset$ and one of the following holds:

1. $A \subseteq \Sigma_1 \setminus \Sigma_2$ and $s \xrightarrow{A}_1 s'$,
2. $A \subseteq \Sigma_2 \setminus \Sigma_1$ and $s \xrightarrow{A}_2 s'$, or
3. $A \subseteq \Sigma_1 \cap \Sigma_2$ and $s \xrightarrow{A}_1 s'$ and $s \xrightarrow{A}_2 s'$.

In other words, components must synchronize on shared actions and proceed independently on local actions. This notion of parallel composition is similar to the definition used for CSP; see also [44].

1.3.4 Transforming PHP into an LKS

We assume that there is a set of services Σ, each with its own implementation. The programs are assumed not to have explicitly generated threads. Instead, we assume that n^σ identical copies of the service $\sigma \in \Sigma$ are running in parallel.

For the verification of the web-service we first construct a formal model for it. We use LKSs for modeling the processes involved in the service. If the source code of a component is not available, e.g., in the case of a third-party service, we assume that an LKS summarizing the interface of the service is written manually, using a (possibly informal) specification of the service as a guide.

If the source code of the component is available, we compute a formal model of the program automatically. The first step is to parse and type-check the PHP program. Our scanner and parser is based on the scanner and parser of the Zend Engine, version 2.0.[3] The Zend engine is also used by most execution environments for PHP.

The type-checking phase is complicated by the fact that the PHP language is typically interpreted, and thus, variables in PHP scripts may have multiple types, to be determined at run-time. We address this problem by introducing

[3] The Zend engine is available freely at `http://www.zend.com/`

multiple 'versions' of each variable, one for each type that the variable might have. A new variable is added that stores the actual type that the program variable has at a given point in time.

The next step is to further pre-process the program. Object construction and destruction is replaced by corresponding calls to the construction and destruction methods, respectively[4]. Side effects are removed by syntactic transformations, the control flow statements (`if`, `while` and so on) are transformed into guarded `goto` statements. As PHP permits method overloading, we perform a standard value-set analysis in order to obtain the set of methods that a method call may resolve to. This is identical to the way function pointers are handled in a programming language such as ANSI-C. The guarded `goto`-program is essentially an annotated control flow graph (CFG). The CFG is then transformed into an LKS, which is straight-forward.

We formalize the semantics of a fragment of PHP using an LKS M^σ for each service $\sigma \in \Sigma$. The behavior of the whole system is given by the parallel composition $M_1^\sigma || \ldots || M_{n^\sigma}^\sigma$ for all services σ, i.e., all copies of all services are composed.

The only point of synchronization of processes is assumed to be a call using SOAP or the like. For each thread i of service σ, we define a synchronization event ω_i^σ. We also define local actions τ_i^σ for all $\sigma \in \Sigma$ and $i \in \{1, \ldots, n^\sigma\}$. The τ_i^σ events are used exclusively by thread i of service σ. If the thread is clear from the context, we simply write $s \xrightarrow{\tau} s'$ for a local transition of the tread.

Notation

The global state space $S = S_1 = \ldots = S_n$ is spanned by the data and variables for all services and a program counter PC_i^σ for each thread. Thus, a state $s \in S$ is a pair $(\overline{V}, \overline{PC})$ consisting of a vector $\overline{V}$ for the program variables and a vector $\overline{PC}$ for the PCs. Given a state $s \in S$, we denote the projection of the value of PC_i from s as $s.PC_i$.

The execution of a statement by thread i increases the PC of thread i, while the other PCs remain unchanged. Let $\nu_i(\overline{\sigma})$ be a shorthand for $\overline{PC}'$ with $PC_i' = PC_i + 1$ and $PC_j' = PC_j$ for all $j \neq i$.

Initialization

We define the set of initial states *Init* as the set of states $s \in S$ such that the PCs are set to the start of each thread. The initialization of the variables is assumed to be performed as part of the program.

Transition Relation

The transition relation of LKS M_i is defined by a case split on the instruction that is to be executed. There are four different statements: assignments,

[4] In the case of PHP, only a very limited form of destruction is performed.

guarded gotos, requests and acknowledgments. Assignments are used to model reading and writing of program variables and transactions on data in the database. The guarded gotos model the control flow of the program. The synchronizing calls to other web services, e.g., by means of SOAP as described above, are modeled by means of two sync statements: the first sync statement is used for the synchronization of the request, and the second sync statement is used for the synchronization of the acknowledgment. The semantics of both statements is identical.

Formally, let $\mathcal{P}_t(PC)$ denote the instruction pointed to by PC in thread t. Let I be a shorthand for $\mathcal{P}_t(s.PC_t)$.

- If I is a $\texttt{sync}(\sigma)$ statement, the thread t non-deterministically chooses a server thread $i \in \{1, \ldots, n^\sigma\}$ and makes a ω_i^σ-transition. No synchronization with other threads is performed. Formally,

$$I = \texttt{sync}(\sigma)\texttt{;} \Longrightarrow s \xrightarrow{\omega_i^\sigma}_t s'$$

  with $s'.\overline{V} = s.\overline{V}$, $s'.\overline{PC} = \nu_i(s.\overline{PC})$.
- If I is a statement that assigns the value of the expression e to the variable x, the thread i makes a τ-transition and changes the global state accordingly. Let $s(e)$ denote the value of the expression e evaluated in state s.

$$I = \texttt{x=e;} \Longrightarrow s \xrightarrow{\tau}_i s'$$

  with $s'.x = s(e)$, $s'.y = s.y$ for $y \neq x$, $s'.\overline{PC} = \nu_i(s.\overline{PC})$. If the modification of x triggers events that other threads are sensitive to, this can be realized by an implicit `notify` statement after the assignment.
- If I is a guarded `goto` statement with guard g and target t, the thread i makes a τ-transition and changes its PC accordingly:

$$I = \texttt{if(g) goto t;} \Longrightarrow s \xrightarrow{\tau}_i s'$$

  with $s'.\overline{V} = s.\overline{V}$, and

$$s'.PC_j = \begin{cases} t & : i = j \ \wedge \ s(g) \\ PC_j + 1 & : \text{otherwise} \end{cases}$$

1.4 Model Checking with Abstraction for Web-Services

1.4.1 Existential Abstraction of Transition Systems with Events

For the verification of the web-service we first construct an abstract, formal model for it. We assume that we have generated or written concrete LKSs as described in Sec. 1.3.4. The concrete LKSs are then abstracted into abstract LKSs. The labels on the states of the abstract LKSs correspond to predicates that are used for predicate abstraction. As done in [45], we use SAT in order

to compute the abstraction of the instructions in the PHP programs. This section provides a short overview of the algorithm. For more information on the algorithm, we refer the reader to [46, 45].

Recall that S denotes the (global) set of concrete states. Let $\alpha(s)$ with $s \in S$ denote the abstraction function. The abstract model makes an A-transition from an abstract state $\hat{s}$ to $\hat{s}'$ iff there is an A-transition from s to s' in the concrete model and s is abstracted to $\hat{s}$ and s' is abstracted to $\hat{s}'$. Let $\hat{T}$ denote this abstract transition relation. Formally,

$$\hat{s} \xrightarrow{A} \hat{s}' : \Longleftrightarrow \begin{array}{l} \exists s, s' \in S : s \xrightarrow{A} s' \wedge \\ \alpha(s) = \hat{s} \wedge \alpha(s') = \hat{s}' \end{array} \tag{1.6}$$

This formula is transformed into conjunctive normal form (CNF) by replacing the bit-vector arithmetic operators by arithmetic circuits. Due to the quantification over the abstract states this corresponds to an all-SAT instance. For efficiency, one over-approximates $\hat{T}$ by partitioning the predicates into clusters [47]. The use of SAT for this kind of abstraction was first proposed in [48]. We use Boolean programs [10] to represent the abstract models. In order to represent the synchronizing events, a special `sync` instruction is added to the language.

1.4.2 Thread-modular Abstraction

The abstract models are built separately for each LKS corresponding to an individual PHP program, or thread of execution. The advantage of this approach is that the individual programs are much smaller than the overall system description, which usually consists of multiple PHP scripts. After abstracting the threads separately, we form the parallel composition of the abstract LKSs, which can then be verified.

The following formalizes our modular abstraction approach.

Let M_1 and M_2 be two LKSs, and let $\pi = \langle s_1, a_1, \ldots \rangle$ be an alternating infinite sequence of states and actions of $M_1 \parallel M_2$. The *projection* $\pi{\restriction}M_i$ of π on M_i consists of the (possibly finite) subsequence of $\langle s_1, a_1, \ldots \rangle$ obtained by simply removing all pairs $\langle a_j, s_{j+1} \rangle$ for which $a_j \notin \Sigma_i$. In other words, we keep from π only those states that belong to M_i, and excise any transition labeled with an action not in M_i's alphabet.

We now record the following claim, which extends similar standard results for the process algebra CSP [49] and LKSs [30].

Claim.

1. Parallel composition is (well-defined and) associative and commutative up to $\sim$-equivalence. Thus, in particular, no bracketing is required when combining more than two LKSs.

2. Let $\hat{M}_i$ denote the abstraction of M_i, and let $\hat{M}_{||}$ denote the abstraction of the parallel composition of $M_1, \ldots, M_n$. Then $\hat{M}_1 || \ldots || \hat{M}_n \sim \hat{M}_{||}$. In other words, the composition of the abstract machines $(\hat{M}_1, \ldots, \hat{M}_n)$ is an abstraction of the composition of the concrete machines $(M_1, \ldots, M_n)$.

For detailed related proofs of the compositional approach, we refer the reader to [49, Chapter 2].

Claim 1 formalizes our thread-modular approach to abstraction. Simulation and refinement can also be performed without building the transition relation of the product machine. This is justified by the fact that the program visible state variables ($\overline{V}$ and $\overline{PC}$) are only changed by one thread on shared transitions. Thus, abstraction, counterexample validation and abstraction refinement can be conducted one thread at a time.

1.4.3 Abstraction-refinement Loop

Once the abstract model is constructed, it is passed to the model checker for the consistency check against the properties. We use the SATABS model checker [14], which implements the SAT-based predicate abstraction approach for verification of ANSI-C programs. It employs a full counter-example guided abstraction refinement verification approach. Following the abstraction-refinement loop, we iteratively refine the abstract model of the PHP program if it is detected that the counterexample produced by the model checker can not be simulated on the original program. Since spurious counterexamples are caused by existential abstraction and since SAT solvers are used to construct the abstract models, we also use SAT for the simulation of the counterexamples. Our verification tool forms a SAT instance for each transition in the abstract error trace. If it is found to be unsatisfiable, it is concluded that the transition is spurious. As described in [50], the tool then uses the unsatisfiable core of the SAT instance for efficient refinement of the abstract model.

Clearly, the absence of individual spurious transitions does not guarantee that the error trace is real. Thus, our model checker forms another SAT instance. It corresponds to Bounded Model Checking (BMC) [51] on the original PHP program following the control flow and thread schedule given by the abstract error trace. If satisfiable, our tool builds an error trace from the satisfying assignment, which shows the path to the error. A similar approach is used in [52] for DSP software. The counterexample is mapped back to the program locations and syntax of the original PHP program in order to provide a useful basis for error diagnosis. In particular, the counterexample trace includes values for all concrete variables that are assigned on the path. If unsatisfiable, the abstract model is refined by adding predicates using weakest preconditions. Again, we use the unsatisfiable core in order to select appropriate predicates.

1.4.4 Object References and Dynamic Memory Allocation

The PHP language is based on C and C++, and thus, makes frequent use of dynamically allocated objects using the `new` operator. Also, it permits to take the address of variables for building references. We support such constructs by the following means:

- We allow references and the (implicit) dereferencing operators within the predicates.
- For each variable that may be assigned a reference, we have special predicates that keep track of the size and a valid bit to track whether the reference is NULL. It is set or cleared upon assignment. Each time the pointer is dereferenced, we assert that the valid predicate holds. We denote the predicate by $\zeta(o)$, for any object o.
- During the construction of Equation (1.6), we employ a standard, but control flow-sensitive points-to analysis in order to obtain the set of variables a pointer may point to. This is used to perform a case-split in order to replace the pointer dereferencing operators. Dynamic objects are handled as follows: we generate exactly as many instances as there are different points that may alias to the same dynamic object.

This approach not only allows handing references within PHP programs, but also efficiently manages the size of the generated CNF equations since it avoids handling data that pointers do not point to.

Note that concurrency issues can be ignored during the alias analysis, as references cannot be shared (practically) among processes. Thus, we can use efficient and precise alias analysis algorithms for sequential programs.

Example

```
class s {
      var $n;
      var $i;
}
...
$p=new s;           ζ(*p)
$p->n=new s;        ζ(*p), ζ(*(p->n))
$p->n->i=$p->i+1;   p->n->i = p->i + 1
```

Fig. 1.7. Example of Abstraction in Presence of Dynamic Objects.

Figure 1.7 shows an example of predicate abstraction in the presence of dynamically allocated objects. The left hand side shows the code to be abstracted, the right hand side shows the predicates that hold after the execution of the code. In order to show the last predicate, the equality of the two integer fields, the following formula is built, where D_1 and D_2 denote the two

dynamic objects, and b_3 denotes the Boolean variable corresponding to the predicate $p\texttt{->}n\texttt{->}i = p\texttt{->}i + 1$:

$$\begin{array}{c} p = \&D_1 \ \wedge \ D_1.n = \&D_2 \ \wedge \\ D_2'.p = D_2.p \ \wedge \ D_2'.i = D_1.i \ \wedge \\ (b_3 \iff (D_2'.i = D_1.i + 1)) \end{array}$$

This formula is only valid for $b_3 = \mathsf{true}$, which shows the predicate.

1.4.5 Case Study

We have experimented with a set of PHP scripts in order to quantify the performance of predicate abstraction on programs written in a scripting language. Two different scripts implement the two parts of the service:

1. A front-end script handles client connections and interacts with them by means of HTML-forms. It maintains the user sessions and uses SOAP to communicate with the back-end script.
2. The back-end script receives commands from the front-end script via SOAP and stores transactions in a MySQL database.

The front-end and back-end scripts have about 4000 and 3000 lines of code, respectively, not including in-line HTML. The property we check is an assertion on the transaction records generated. It is enforced by the front-end script (input data not compliant is rejected). For the purpose of verification, we add an assertion that checks it in the back-end as well. We verify this property for an unbounded number of client and front-end processes, and one back-end process.

The main challenge when applying formal methods to scripting languages such as PHP is to model the extensive library (as implemented by the PHP execution environment). We have only completed that task to the point that was required for the property described above to pass; a verification tool of industrial relevance has to include an almost complete set of models for all functions offered by the execution environment. Similar issues exist for languages such as JAVA and C# as well, however. In particular, the direct access to the SQL databases permitted by the PHP scripting language actually requires statically analyzing SQL commands. Unfortunately, PHP does not provide a suitable abstraction layer, and thus, the commands used to access databases even depend on the vendor.

Our verification engine has been applied in the past to ANSI-C programs of much larger size and higher complexity, and thus, typical scripts do not pose a capacity challenge. The verification requires only 5 refinement iterations, generating 120 predicates, and terminates within 51 seconds on a modern machine with 3.0 GHz. Most components of the abstraction-refinement loop have linear run-time in the size of the programs. The only exception is the verification of the abstract model, which may be exponential in practice.

However, in the case of PHP, this is rarely observed: as there is very little interaction between the processes (when compared with C programs that use shared-variable concurrency), the partial order reduction that our model checker implements eliminates almost all interleavings between the threads, and the complexity of verifying the abstract models becomes comparable to that of checking sequential programs.

1.5 Conclusion

This paper formalizes the semantics of a PHP-like language for implementing web services by means of labeled Kripke structures (LKSs). The LKS notation permits both synchronizing and non-synchronizing (interleaving) communication in the model. Both forms of communication are typical for web services. While each form of communication can be replaced by the other one, doing so typically results in a blowup of the model. The LKSs of the threads can be analyzed formally by means of automatic predicate abstraction. These results enable the verification of the applications that implement web services.

We have implemented these techniques in a tool called SatAbs. While our implementation is currently limited to the verification of PHP5 scripts, the method is also applicable to other programming languages used in this context. It could be extended to handle systems that use multiple programming languages, e.g., both PHP5 and JAVA, or PHP5 and C#.

Our implementation is able to show safety properties of a combination of multiple PHP scripts running in parallel. Most steps of the analysis loop are done in a thread-modular manner, and thus, the analysis scales in the number of threads. The verification of the abstract (Boolean) model is the only part of the analysis that examines the entire system.

Our implementation currently lacks support for liveness properties, despite of the fact that liveness is a property of high importance in the context of web services. While predicate abstraction is in general suitable to prove liveness properties, it has to be augmented with a generator for ranking functions in order to prove termination of most loops [53]. Another future direction of research is to extend the implementation to prove concealment properties, e.g., that session IDs do not leak out.

Model checking for PHP or similar scripting languages possibly has applications beyond property checking. For example, the information obtained about the reachable state-space could be exploited to efficiently compile a PHP program (which is usually interpreted) into machine code.

References

1. D.K. Barry. Web services and service-oriented architectures. *Morgan Kaufmann*, 2003.
2. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, December 1999.
3. E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*. Springer, 1981.
4. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
5. Gerard J. Holzmann and Doron Peled. An improvement in formal verification. In Dieter Hogrefe and Stefan Leue, editors, *Formal Description Techniques VII, Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques, Berne, Switzerland, 1994*, volume 6 of *IFIP Conference Proceedings*, pages 197–211. Chapman & Hall, 1995.
6. Patrik Cousot. Abstract interpretation. *Symposium on Models of Programming Languages and Computation, ACM Computing Surveys*, 28(2):324–328, June 1996.
7. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer, 1997.
8. M. Colón and T.E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Computer Aided Verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science*, pages 293–304. Springer, 1998.
9. E. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *POPL*, 1992.
10. T. Ball and S.K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report 2000-14, Microsoft Research, February 2000.
11. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Language Design and Implementation*, pages 203–213. ACM, 2001.
12. Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *ICSE*, pages 385–395. IEEE Computer Society, 2003.

13. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with BLAST. In Thomas Ball and Sriram K. Rajamani, editors, *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer, 2003.
14. Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicated abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference (TACAS)*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer, 2005.
15. PHP: Hypertext preprocessor. `http://www.php.net/`.
16. `http://www.asp.net/`.
17. http://www.ibm.com/developerworks/library/specification/ws-bpel/.
18. Bernd-Holger Schlingloff, Axel Martens, and Karsten Schmidt. Modeling and model checking web services. *Electr. Notes Theor. Comput. Sci.*, 126:3–26, 2005.
19. Raman Kazhamiakin, Marco Pistore, and Luca Santuari. Analysis of communication models in web service compositions. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 267–276, New York, NY, USA, 2006. ACM Press.
20. Marco Pistore, Paolo Traverso, Piergiorgio Bertoli, and Annapaola Marconi. Automated synthesis of executable web service compositions from BPEL4WS processes. In Ellis and Hagino [21], pages 1186–1187.
21. Allan Ellis and Tatsuya Hagino, editors. *Proceedings of the 14^{th} international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005 - Special interest tracks and posters*. ACM, 2005.
22. Tuba Yavuz-Kahveci, Constantinos Bartzis, and Tevfik Bultan. Action language verifier, extended. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 413–417. Springer, 2005.
23. Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of interacting BPEL web services. In Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, editors, *WWW*, pages 621–630. ACM, 2004.
24. Xiang Fu, Tevfik Bultan, and Jianwen Su. Model checking XML manipulating software. In George S. Avrunin and Gregg Rothermel, editors, *ISSTA*, pages 252–262. ACM, 2004.
25. Tevfik Bultan, Xiang Fu, Richard Hull, and Jianwen Su. Conversation specification: a new approach to design and analysis of e-service composition. In *WWW*, pages 403–410, 2003.
26. Daniel Kroening and Natasha Sharygina. Formal verification of SystemC by automatic hardware/software partitioning. In *Proceedings of MEMOCODE 2005*, pages 101–110. IEEE, 2005.
27. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.
28. Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
29. Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1):1–40,41–77, September 1992.
30. E. Clarke, S. Chaki, N. Sharygina, J. Ouaknine, and N. Sinha. State/event-based software model checking. In *Proceedings of the International Conf. on Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*. Springer, 2004.

31. R. De Nicola and F. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM (JACM)*, 42(2):458–487, 1995.
32. Ekkart Kindler and Tobias Vesper. ESTL: A temporal logic for events and states. In *Application and Theory of Petri Nets 1998, 19th International Conference (ICATPN'98)*, volume 1420 of *Lecture Notes in Computer Science*, pages 365–383. Springer, 1998.
33. M. Huth, R. Jagadeesan, and D. Schimidt. Modal transition systems: A foundation for three-valued program analysis. In *Lecture Notes in Computer Science*, volume 2028, page 155. Springer, 2001.
34. Shin Nakajima. Model-checking of safety and security aspects in web service flows. In Nora Koch, Piero Fraternali, and Martin Wirsing, editors, *Web Engineering - 4th International Conference, ICWE 2004, Munich, Germany, July 26-30, 2004, Proceedings*, volume 3140 of *Lecture Notes in Computer Science*, pages 488–501. Springer, 2004.
35. Marco Pistore, Marco Roveri, and Paolo Busetta. Requirements-driven verification of web services. *Electr. Notes Theor. Comput. Sci.*, 105:95–108, 2004.
36. Alin Deutsch, Monica Marcus, Liying Sui, Victor Vianu, and Dayou Zhou. A verifier for interactive, data-driven web applications. In Fatma Ozcan, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 539–550. ACM, 2005.
37. Xiang Fu, Tevfik Bultan, and Jianwen Su. Realizability of conversation protocols with message contents. In *ICWS*, pages 96–. IEEE Computer Society, 2004.
38. Howard Foster, Sebastián Uchitel, Jeff Magee, and Jeff Kramer. Compatibility verification for web service choreography. In *ICWS*, pages 738–741. IEEE Computer Society, 2004.
39. Aysu Betin-Can, Tevfik Bultan, and Xiang Fu. Design for verification for asynchronously communicating web services. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 750–759, New York, NY, USA, 2005. ACM Press.
40. Dirk Beyer, Arindam Chakrabarti, and Thomas A. Henzinger. Web service interfaces. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 148–159, New York, NY, USA, 2005. ACM Press.
41. D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. Web services architecture. World-Wide-Web Consortium (W3C). Available from http://www.w3.org/TR/ws-arch/, 2003.
42. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, January 2003.
43. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.
44. T. S. Anantharaman, E. M. Clarke, M. J. Foster, and B. Mishra. Compiling path expressions into VLSI circuits. In *Proceedings of POPL*, pages 191–204, 1985.
45. Himanshu Jain, Edmund Clarke, and Daniel Kroening. Verification of SpecC and Verilog using predicate abstraction. In *Proceedings of MEMOCODE 2004*, pages 7–16. IEEE, 2004.
46. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design*, 25:105–127, September–November 2004.

47. Himanshu Jain, Daniel Kroening, Natasha Sharygina, and Edmund Clarke. Word level predicate abstraction and refinement for verifying RTL Verilog. In *Proceedings of DAC 2005*, pages 445–450. ACM, 2005.
48. Edmund Clarke, Orna Grumberg, Muralidhar Talupur, and Dong Wang. High level verification of control intensive systems using predicate abstraction. In *First ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE'03)*, pages 55–64. IEEE, 2003.
49. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall International, London, 1997.
50. E. Clarke, H. Jain, and D. Kroening. Predicate Abstraction and Refinement Techniques for Verifying Verilog. Technical Report CMU-CS-04-139, 2004.
51. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Yhu. Symbolic model checking without BDDs. In *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.
52. David W. Currie, Alan J. Hu, and Sreeranga Rajan. Automatic formal verification of DSP software. In *Proceedings of DAC 2000*, pages 130–135. ACM Press, 2000.
53. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond safety. In *Computer Aided Verification, 18th International Conference, (CAV)*, volume 4144 of *Lecture Notes in Computer Science*, pages 415–418. Springer, 2006.