An Automata-Theoretic Approach to Open Actor System Verification

Ilham W. Kurnia

Vom Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades

Doktor der Naturwissenschaften (Dr. rer. nat.)

genehmigte

Dissertation

Datum der wissenschaftlichen Aussprache: 23. Januar 2015

Dekan:	Prof. Dr. Klaus Schneider
Vorsitzender der Promotionskommission:	Prof. Dr. Jens Schmitt
Erster Berichterstatter:	Prof. Dr. Arnd Poetzsch-Heffter
Zweiter Berichterstatter:	Prof. Dr. Roland Meyer

D 386

To Mom

Abstract

Open distributed systems are a class of distributed systems where

- only partial information about the environment, in which they are running, is present,
- new resources may become available at runtime, and
- a subsystem may become aware of other subsystems after some interaction.

Modeling and implementing such systems correctly is a complex task due to the openness and the dynamicity aspects. One way to ensure that the resulting systems behave correctly is to utilize formal verification.

Formal verification requires an adequate semantic model of the implementation, a specification of the desired behavior, and a reasoning technique. The actor model is a semantic model that captures the challenging aspects of open distributed systems by utilizing actors as universal primitives to represent system entities and allowing them to create new actors and to communicate by sending directed messages as reply to received messages. To enable *compositional* reasoning, where the reasoning task is reduced to independent verification of the system parts, semantic entities at a higher level of abstraction than actors are needed.

This thesis proposes an automaton model and combines sound reasoning techniques to compositionally verify implementations of open actor systems. Based on I/O automata, the model allows automata to be created dynamically and captures dynamic changes in communication patterns. Each automaton represents either an actor or a group of actors. The specification of the desired behavior is given constructively as an automaton. As the basis for compositionality, we formalize a component notion based on the *static structure* of the implementation instead of the *dynamic* entities (the actors) occurring in the system execution. The reasoning proceeds in two stages. The first stage establishes the connection between the automata representing single actors and their implementation description by means of weakest liberal preconditions. The second stage employs this result as the basis for verifying whether a component specification is satisfied. The verification is done by building a simulation relation from the automaton representing the implementation to the component's automaton. Finally, we validate the compositional verification approach through a number of examples by proving correctness of their actor implementations with respect to system specifications.

Acknowledgments

First of all I would like to thank my advisor, Prof. Dr. Arnd Poetzsch-Heffter, for his continuous support, from caring a lot about my work to ensuring that all administrative and bureaucratic obstacles were resolved in advance. The opportunity he has given me to pursue my doctorate in Kaiserslautern has enabled me to grow not only academically but also in other aspects of life. I also thank Prof. Dr. Roland Meyer who reviewed this thesis as the second reviewer and Prof. Dr. Jens Schmitt for chairing the doctoral committee.

In the course of my study, I had the luxury to be working in the EU Project HATS, led by Prof. Dr. Reiner Hähnle. The project brought plenty of interaction, new acquaintances, traveling and work experience, and many other positive aspects. It is a great pleasure to take part in advancing practical formal verification a step further.

This dissertation could not be brought to fruition without the interaction I had within the Software Technology working group at University of Kaiserslautern. I am especially grateful to Annette Bieniusa, Christoph Feller, Jan Schäfer and Yannick Welsch for allowing me discuss research problems in depth, sometimes under time pressure. The encouragement Ina Schaefer once said to me: “never give up” kept me going at rough times. And to my ex-office mate, Kathrin Geilmann, thank you for enduring my yapping and often ridiculous questions about Germany and the German language.

The quality of the dissertation would have taken a severe hit, if it were not for the detailed proofreading by Annette, Christoph, Yannick, Deepthi Akkoorath, Peter Zeller and Carroline Dewi Puspa Kencana Ramli. The administrative and technical support from Judith Stengel, Gabriele Sakdapolrak, Thomas Schneider and Bernd Schurmann, among others is deeply appreciated.

My university life would not be complete without the social support I was fortunate to have. To ISGS, Ania, Christiano, Dasha, Misha, Olga, Paddy, Ramy, Viktor: thank you for being there when I needed it the most.

Last but not least, I would like to thank my parents and my brother and his family. Without their constant support, I could not see myself going for my Ph.D. to begin with.

Contents

List of Figures

List of Tables

List of Definitions, Theorems, and Lemmas

Definitions

Theorems

Lemmas

CHAPTER 1

Introduction

The current state of technology dictates that distributed systems are integrated into many daily activities, such as controlling modern household appliances and performing valid transactions in the field of e-commerce. They are often designed to run continuously. Combined with the never-ending development of technology, they need to be able to cope with operating in and interacting with some not fully understood environment (e.g., the Internet). Systems that are designed to handle this condition are called *open distributed systems*.

Developing open distributed systems is a non-trivial task. Various independent components of these systems are typically executed on various kinds of machines such as desktop computers, servers and mobile devices. Communication between components is affected by the underlying network whose structure may change over time. The systems themselves tend to be dynamic, allowing components to be added, replaced, or removed with little control of the environment. Because of their importance and complexity, the International Organization for Standardization, the International Electrotechnical Commission, and the Telecommunication Standardization Sector created a reference model for open distributed systems [IT95].

Actor model. One established programming model that caters for this reference model is the *actor model* [HBS73; Agh86]. The actor model, which is adopted by prominent programming languages for building distributed systems such as Erlang [Arm03] and Scala [OSV11], follows the object modeling approach recommended in the reference model. As in the object-oriented paradigm, the actor model views a system as a collection of uniquely named *actors*. Each actor is usually seen as an object that has an *encapsulated* state and a single-threaded processor, meaning that an actor has exclusive control over its state and processor that other actors cannot influence directly. Actors communicate exclusively with each other via *asynchronous message passing* addressed with the actor's name. Actors react to incoming messages by possibly changing their state, creating more actors, and sending out messages to other actors according to some code asso-

ciated with them. Because of the encapsulation the actor model abstracts from the issues pertinent to the actual conditions on which the actors execute, such as the type of processors, their speed and the underlying network systems. The communication mechanism lends itself as a simple, scalable communication interaction model in a distributed setting [CH05, Chapter 1]. Being *input-enabled* (i.e. accepting all messages addressed to it [Lyn96, p. 257]), an actor is by design open. Actor creation and sending the names of other actors within the messages represent the dynamicity of the open distributed systems.

The actor model succeeds in providing a high-level abstraction of open distributed systems. Ensuring such a system operates *correctly* is still nevertheless a difficult problem ([CY83; Roe+01, Chapter 1]). With errors in software systems in general proving to be very costly[1], it is desirable to perform the correctness check in a systematic and reliable manner. Testing, a successful technique to improve the quality of systems running sequentially on a single machine, is much less appropriate for open distributed systems because of its high degree of non-determinism. Formal verification has been pushed as the proper method to perform this task (see, e.g., [BK08]).

Verification context. Formal verification is a methodical, mathematical-based analysis to prove that a system satisfies a formal description (i.e., *specification*) of the desired properties. To perform this analysis, the system needs to be modeled as a mathematical entity such that it can be verified against the specification. This mathematical entity needs to be chosen carefully to establish a strong connection with the implementation. The way the specification is described and the system is verified also influences the choice of the entity.

A formal verification technique for a category of systems makes use of prominent characteristics of the category in order for the analysis to be effective. For actor systems we identify the following characteristics inherited from the open distributed systems:

- The heterogeneity of the underlying infrastructure implies that concurrent computations may progress at different speed.
- The dynamicity of the underlying network infrastructure implies that messages take arbitrary time to be sent and may arrive out of order.
- Actor names can be passed around and new actors can be created, allowing new connections to appear. The topology of the network built by the actor

[1] The University of Cambridge estimated that the global cost of errors in software is 312 billion USD. http://www.prweb.com/releases/2013/1/prweb10298185.htm

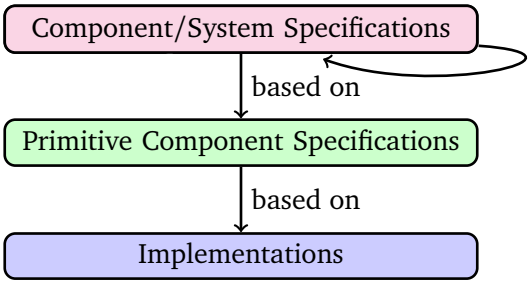


Figure 1.1.: Compositional verification

system is dynamic.

- Being open, an actor system has little control of its environment. This means that once an actor is *exposed* to the environment, the environment remembers the knowledge of this actor forever.

Verifying actor systems. Distributed systems tend to be complex, with their development done on a component basis. Similarly, the verification of an actor system is ideally done in a *compositional* (also known as *modular*) manner [MC81; Zwi89]. This compositional principle works in the way that the specification of a component is verified on the basis of the specification of subcomponents that have been previously verified. The specification represents the desired *black-box* or *functional* behavior of the component. The implementation of an indivisible or *atomic* or *primitive* component is verified against the specification, as illustrated by Figure 1.1. The final goal is to verify the whole system. With the compositional principle in place, verifying that the system satisfies desired properties can be done only by dealing with the specification of its constituents.

To apply this verification approach, the following three questions need to be answered.

Q1. What are components in the context of verification of actor-based systems?

Q2. How do we specify the desirable properties?

Q3. Which verification technique is suitable for performing the compositional verification?

The answer of the first question is strongly linked to how we model the implementation. In turn, this affects the choice of specification and verification techniques. The following sections dissect the three questions above in more detail, presenting an overview of this thesis.

1.1 Implementations and Components

There is no universal answer to the question what components are, as the precise notion of components depends strongly on the underlying system implementation and the goal that we want to achieve with the presence of components. In this thesis, we want to use components as entities that capture independent parts of the implementation of an actor system. The desired characteristics of a software component as defined by Szyperski [Szy98] are: stateless, composable by third parties, independently deployable and providing a specific interface. To come up with a definition of component that fulfills these characteristics, we first need to discuss what kind of implementation language we use to describe the behavior of the actors.

In general, there are two approaches how to program the behavior of actors:[2]

- the *update*-based approach [HBS73; Agh86; Lie87; Arm10] and
- the *class*-based approach [YBS86; VA01; SJ11; JOY06; JHSSS11].

In the update-based approach, the behavior of an actor is described by a parameterized procedure. The procedure starts with a pattern matching construct to filter what messages the actor can process. Then follows the desired behavior of the actor which may send messages and create new actors. At the end of the procedure, the description of the behavior of the actor is *replaced* by a potentially different procedure. As a result, the actor may now behave totally differently from before, whether it is the set of messages it can process, the messages it produces, the new actors it creates and so on. The class-based approach advocates for a more static description, where the behavior of an actor is fixed to a class. The class contains a number of method definitions, each describing a message pattern the actor can process, and a number of fields describing the internal state of the actor. Each method definition describes the messages the actor sends, the new actors that are created, and the changes in the internal state.

The class-based approach is the approach chosen in this thesis as motivated by the following reasons:

- The class-based approach is closer to the object-oriented paradigm hailed by the reference model [IT95].
- The class-based approach has more ingredients to provide a *type-safe* environment [Pie02] for verification, so that orthogonal issues such as checking

[2] In a way, these two approaches resemble the two popular description mechanisms for objects: the prototype-based approach — first introduced in Self [US87] and later adopted by JavaScript [ECM11] — and the class-based approach – first introduced in Simula [DN66] and later adopted by C++ and Java.

whether a message can ever be processed by an actor (i.e., fits a pattern) can be solved statically through the use of static type checkers.

- It provides *static* names to refer to a group of actors that have the same set of possible behaviors.

Furthermore, the class-based approach supports the methodology of *programming to interfaces* [GHJV95]. An agreement over the interfaces allows the users of the interfaces to abstract from the irrelevant details that are used by their providers, which in turn reduces the complexity of developing software in general.

The language we use as the basis for the implementation is a simplified version of ABS [JHSSS11], short for Abstract Behavioral Specification language. This class-based language is designed as a middle ground between

- design-oriented and architectural languages such as UML [RJB04],
- minimal executable formalisms for specifying concurrent and distributed behavior such as CSP [Hoa78] and π-calculus [Mil99], and
- implementation-oriented specification languages, such as JML [LC06] for Java [GJJ96] that applies the 'design by contract' principle [Mey97].

Its syntax adapts the syntax of Java, focusing only on the core features needed for describing the desired behavior of actors. Realizing the advantage of being an abstract language [Pla13], the simplicity of ABS allows the development of automatic translators to Java, Maude ([Cla+07]), Scala, and Haskell ([Jon03]). The work on the automatic translators are reported, e.g., in [AØVWW13]. ABS features *futures* [BH77; Hal85; LS88]: a construct to synchronize a method call and its return result. By using futures, the request/reply communication pattern that appears very often in actor systems [HO09] can be elegantly solved without having to expose the caller's name and introducing a new process on the caller's side to act upon the return.

Having decided on the class-based approach, the notion of components should be class-based as well, following the component characteristics defined by Szyperski. An actor on its own is an independent executable entity with a clear interface. Thus, its static behavioral representation (i.e., its class) is fit to become the primitive component. One characteristic that can be used to define the general notion of components is the creation relation between the actors. This relation can be deduced from the class descriptions. A set of classes is said to be *creation-complete* if an actor of a class within the set creates only actors whose classes are also within the set [KPH13]. Combining a creation-complete set of classes with one specific *activator class* in the set, following the basic idea from software component models such as OSGi [Osg] and COM [Mic99], yields our *components*. The *component instances* are groups of actors, each of which consists of an actor of the

activator class and all other actors transitively created by that actor. This instance notion is similar to the actor components defined by Agha, Mason, Smith and Talcott [AMST97], which are *dynamic* entities. Because the behavior description of all actors in these groups is static, we have the basis for statically analyzing groups of actors. The exact members of the component instances are influenced by their interaction with the environment.

A key feature of this component definition is that given an activator class, the corresponding component can be derived by going through each class definition. If a class C is contained in a component derived from an activator class AC, then C is also contained in a component that is derived from another activator class and that includes AC. Because of this hierarchical structure, components whose specifications have been verified can be used as the ingredients to verify other components, allowing compositional verification.

1.2 Automata-Based Specification

After establishing the precise context of the implementations, we now proceed to discuss the specifications. Lamport [Lam83b] distinguishes a specification technique based on two factors:

- The way the specification is described: *constructive* or *axiomatic*.
- What the specification talks about: the *actions* that the implementation should do or the change of *states* the implementation should experience.

A constructive specification "describes an abstract model that tells how the [implementation] should behave". On the opposite side, an axiomatic specification directly states what *properties* the implementation should have. The desire for compositional verification means that the chosen specification technique should be able to tackle the *hierarchical* nature of the components. Furthermore, a specification needs to have a clear, formal semantics, which becomes the basis for proving the soundness of the selected verification method.

Several formalisms and their associated specification techniques have been introduced to model distributed systems, for example, process calculi [Hoa78; CG97; Mil99], Petri nets [Pet62] and temporal logics [Pnu77]. In this thesis, we adopt a constructive, automata-based specification technique. Choosing automata as the semantics for the specifications has the following advantages:

- Automata can incorporate both notions of actions and states in the same model. From a semantical point of view, actions are the atomic units of traces, while

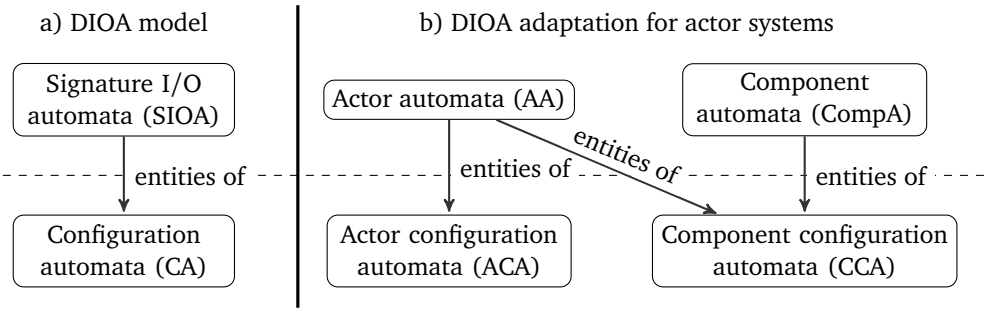


Figure 1.2.: Adaptation of DIOA model for actor systems

traces are typically the basis for a *fully abstract* semantics (see, e.g., for object-oriented setting [JR05; AGGS09; WP14]). A fully abstract semantics captures the *minimal* information needed to define the *observable* behavior without relying on the implementation details [Plo77]. Thus, using actions is ideal to specify the behavior of a system on different levels of abstraction. The use of states can simplify the specification, in particular for specifying the impact of different related actions being sent to an actor or a component instance.

- (Parallel) composition is a standard operator on automata. The correctness of a composed automaton can often be proved through some compositional reasoning.

Many automaton models have been used to formally represent the behavior of distributed systems (see, e.g., [BZ83; LT87; AH01]). Usually these models are static, meaning that the set of states and transitions of the automaton must be decided up front. Modeling dynamic systems such as actor systems and ad-hoc networks where the participating members of the systems may vary is done by the addition of Boolean flags. For example, all possible actors are assumed to be present all along, but only become alive (i.e., the flag changes to true) after special creation events are executed [Leo90; LMWF93]. This approach has a severe disadvantage in an open setting as all possible actors of all possible behavior need to be represented in the automata already from the initial states. Consequently, the supposed advantage of having a composition operator becomes non-existent. To obtain a model that more closely represents actor systems, we follow Attie and Lynch's proposal: the *dynamic I/O automaton* (DIOA) model [AL01; AL15].

The DIOA model is based on I/O automata [LT87], an automaton model that caters well for modeling open systems [FL05]. An I/O automaton is a state transition system whose transitions are labeled with *actions* which are statically ca-

tegorized as input, output and internal actions. The input and output actions are used to communicate with the environment, while the internal actions are used for internal purposes, such as triggering certain side effects to be executed. This categorization is called the *signature* of the automaton.

The DIOA model extends I/O automata by enabling dynamic changes in their signatures (i.e., the set of events the represented entity can participate in) and the creation of other I/O automata. The DIOA model introduces two layers of I/O automata: *signature I/O automata* (SIOA) and *configuration automata* (CA), whose relation is portrayed in Figure 1.2 a). The SIOA, which represent the entities of the systems, extend I/O automata by allowing the signatures to change dynamically according to their own states. The semantics of a system is represented by CA which keep track of the set of created SIOA. A CA[3] is based on the notion of a *configuration* which contains the set of created SIOA and a mapping to keep track of the current state of each of these SIOA. The transitions of a CA are derived *intrinsically* from the SIOA. Each time a transition involves an action that creates new SIOA, the configuration is extended to include these new SIOA.

The DIOA model has been developed as a model for ad-hoc networks and its adaptation to represent object-based settings has been left as an open question [AL15]. To represent actor systems using the DIOA model, the SIOA and the CA are extended to reflect the underlying behavior of the actors. The SIOA are refined to *actor automata* (AA) where the states and the transitions are defined so that each AA represents typical characteristics of an actor. To provide a close connection to the implementation, the specification of the desired AA uses trace-based class invariants. In this way, the specification of the AA is a class specification.

While an AA defines what an actor can do, it is also defined with an open setting in mind. The implication of this definition is that the environment knows all actors, i.e., it is exposed to all actors within an actor system. However, in a typical execution of an actor system, not all created actors are exposed to the environment. At its extreme, only the initial actor is exposed to the environment, while all other actors are not directly callable by the environment. This exposure information changes over time as more actors are exposed to other actors, changing the *communication topology* within the system and between the system and the environment. To correctly model this *dynamic topology* aspect, we extend CA to *actor configuration automata* (ACA) that provide a notion of *actor configurations* which have AA as their members and keep track of exposed actors. The information on these exposed actors allows a more precise derivation of the action signatures, where actions that represent the environment calling non-exposed ac-

[3] We use abbreviations for automata to also represent "a single automaton". The usage is apparent from the context.

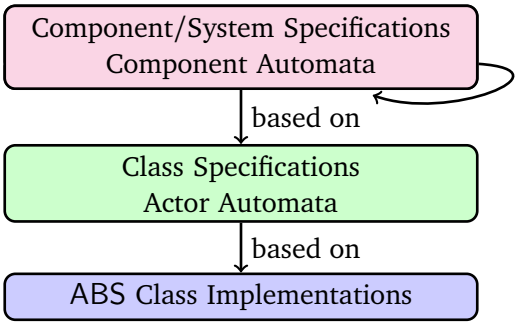


Figure 1.3.: Two-tier verification with DIOA

tors of the system are excluded. The relation between ACA and AA is portrayed in Figure 1.2 b).

With appropriate modifications, the DIOA model can accommodate the component notion. First, we refine the SIOA to *component automata* (CompA) where the states and transitions are defined such that each CompA captures the possible *external* behavior of a component instance. Being refinements of SIOA, CompA and AA share the same fundamental structures. This means that we can compose them together to know how they behave when they execute in parallel.

The ACA do not cover CompA. Therefore, the CA also need to be refined where the component instances have a place in the configuration. That is, CompA can also be part of a configuration. We define *component configuration automata* (CCA) which fulfills this requirement. In fact, a configuration of a CCA contains AA, CompA and also SIOA that represents compositions of a set of AA and/or CompA. Allowing a set of AA, CompA and the composed SIOA to stand on an equal level in the configurations provides the means to perform the verification of a component implementation as we explain below. The introduction of CCA and CompA completes the adaptation of the DIOA model for open component-based actor systems as portrayed in Figure 1.2 b).

1.3 Reasoning Technique

Combining the component notion and the specification technique described in the previous sections, we investigate how to instantiate the general compositional verification method illustrated by Figure 1.1 to Figure 1.3. The approach taken here is divided into two tiers. The first tier is about verifying that a class implementation satisfies its class specification. Once that is established, the verification procedure

proceeds to the second tier, where a component or system specification is verified from the subcomponent specifications. Each tier uses a distinct technique that is appropriate to handle the different complexity present in each tier.

For the first tier, we use a sound, compositional, local verification technique developed by Din et al. [DJO05; DDJO12; DDO12a]. This technique consists of two steps. First, an αABS class implementation is encoded into a simple sequential language SEQ with non-deterministic assignments [Apt84]. This encoding procedure follows a transformational approach originally proposed by Olderog and Apt [OA88]. A SEQ has an established weakest (liberal) precondition semantics ([Dij75]), from which we can use logical inference to verify whether the semantics of the class implementation satisfies a specification for the class in form of class invariants. We show the link between the class invariants and the AA class specifications by comparing them to a trace-based denotational semantics of the classes derived from the work by Ahrendt and Dylla [AD12]. In a nutshell, their relationship in terms of some class C can be characterized as

$$Traces(C_{impl}) \subseteq Traces(\mathcal{A}_C)$$

where C_{impl} represents the implementation of C in αABS and $\mathcal{A}_C$ represents the actor automaton of C.

For the second tier, we adapt the notion of *possibility map* [LT87; NS94], which is first developed for comparing I/O automata. A possibility map is a particular simulation relation ([Par81]) that maps several states s_l of a lower-level automaton to a state s_h of a higher-level automaton. The mapping holds only if s_h can mimic the non-internal transitions that can be taken by s_l and the resulting states of the transitions are in the relation. That is, if s_l makes a non-internal transition t to s'_l, then the transition t can be performed at s_h to go to s'_h, and s'_l and s'_h are related. This notion allows us to conclude that if an SIOA produces fewer traces than another SIOA, then replacing this other SIOA on the CA level implies that no new behavior is introduced. More specifically for the actor setting, the implementation of a component represented by an SIOA composed from AA and CompA representing single actors and subcomponent instances produces no new behavior that is not within its CompA specification. Because we may need to deal with some creation process to represent some internal behavior, the possibility map relates states of two *CCA*: one that uses the composed SIOA and another that uses the CompA. If a possibility map is found, we can conclude

$$Traces(\mathcal{C}_{\{\mathcal{A}_1,\ldots,\mathcal{A}_n\}}) \subseteq Traces(\mathcal{C}_{CompA})$$

where

- $\mathcal{A}_1, \ldots \mathcal{A}_n$ represent the AA and CompA derived from the class specifications and the subcomponent specifications,
- $\mathcal{C}_{\{\mathcal{A}_1,\ldots,\mathcal{A}_n\}}$ represents the CCA of the component implementation, and
- $\mathcal{C}_{CompA}$ represents the CCA of the component specification.

By transitivity, the result above implies

$$Traces(implementation) \subseteq Traces(\mathcal{C}_{CompA}) \ .$$

The sound adaptation of different verification techniques to the two-tier approach becomes a part of the contributions of this thesis, as compiled in the next section.

1.4 Contributions and Outline

This thesis develops an automaton model that allows modular verification of actor-based component systems. The main contributions of this thesis are:

- The development of a component notion for actor systems that enables modular verification, not only on the model level of the specification, but also of the implementation.
- The development of an automaton model based on DIOA that faithfully captures fundamental properties of actor systems: the dynamic creation and the dynamic topology. This model provides an alternative semantical model to developed operational or denotational semantics for actor systems. It also incorporates a representation of futures.
- A compositional specification technique that has a direct translation to the automaton model. This specification technique bridges the action-based and state-based approaches in a suitable way for the actor model.
- A sound adaptation of a trace-based verification technique by Din et al. [DJO05; DDJO12; DDO12a] to reason about the correctness of a class implementation given an automaton specification for the class.
- A possibility map notion for the automaton model to reason about the correctness of system behaviors.

To describe these contributions, the thesis is structured into three parts. Part I, consisting of Chapters 2 to 5, sets up the precise setting by first introducing a running example (Chapter 2). This part motivates the use of the compositional verification framework proposed in Chapter 1 and provides the building blocks

for the automaton model. Chapter 2 starts with an overview of the modeling and verification framework via a client-server example, which is the running example. Then, Chapter 3 formalizes the actor-based language αABS used to illustrate a running example. Based on this αABS description, Chapter 4 describes the basic formalisms necessary to define the automaton framework. This chapter also provides a trace-based denotational semantics employed to prove the soundness of the automaton semantics. Chapter 5 formalizes the component concept.

Part II describes how an automaton model based on the DIOA model (Chapter 6) can be constructed from the setting (Chapters 7 to 9). Chapter 6 summarizes the DIOA framework which consists of two layers: SIOA and CA. Chapters 7 and 8 describe how classes and actor systems can be represented by SIOA and CA, respectively. Chapter 9 describes the adaptation of the component notion defined in Chapter 5 into the DIOA framework. The last chapter of this part, Chapter 10, formalizes how the automata can be specified.

Part III realizes the hierarchical verification model in two steps. First, Chapter 11 describes a sound verification of a class implementation with respect to a class automaton specification. Then, Chapter 12 describes how the component specifications can be verified from other specifications that have been verified. Chapter 13 provides several case studies how the verification model is applied.

Finally, Chapter 14 provides some concluding remarks of the thesis. When appropriate, a discussion is provided at the end of a chapter. Chapter 5 is based on [KPH13] while Parts II and III and are derived from [KPH15]. Before we start with the first part, the following section provides the notational conventions used in this thesis.

1.5 Notation

We rely on the following notations and conventions. Variables are typically written in *italics*. Program(fragment)s are written in a `monospaced font`. Common data types **N** and **B** represent natural numbers and booleans, respectively. We use the common Boolean operator precedence:

- negation $\neg$ binds tighter than conjunction $\wedge$,
- conjunction binds tighter than disjunction $\vee$, and
- conjunction and disjunction have a higher precedence than implication $\Longrightarrow$.

We use four abstract data structures to describe the formal definitions and specifications of functional behavior, namely tuple, sequence, set and map. A tuple is an ordered finite list of elements represented using angled brackets $\langle\rangle$.

The sequence data structure is represented by $Seq\langle T\rangle$, with T denoting the type of the sequence elements. Sequences are typically represented as normal variables (e.g., s) or variables of type T with an overbar (e.g., $\overline{e}$). An empty sequence is denoted by $[\,]$ and $\cdot$ represents sequence concatenation. We also use juxtaposition to represent sequence concatenation when it is clear from the context. The notation s **pr** s' means that a sequence s is a prefix of another sequence s'. The function $Pref(s)$ yields the set of all prefixes of s. The projection operator $s{\downarrow}X$ produces the longest subsequence[4] of the sequence s that contains sequence elements in X. A sequence s_1 is *contained* within another sequence s_2, written $s_1 \subseteq s_2$ if there are s', s'' such that $s' \cdot s_1 \cdot s'' = s_2$. An element e occurs before e' in a sequence s, written $e <_s e'$, if there is s' such that $e \cdot s' \cdot e' \subseteq s$. Note that if e or e' occurs multiple times, this predicate behaves positively by picking suitable specific occurrences of e and e'. The length of a sequence s is denoted by $|s|$. If s is an infinite sequence, then $|s| = \infty$. The n-th element on the sequence is retrieved by $s[n]$ provided $|s| \geq n$. The notation e^* represents a finite sequence containing only of e's.

The set data structure $Set\langle T\rangle$ is a container of values of type T. Set variables are typically written in capital letters (e.g., S). A bold face indicates that the variable is used as a general universe, e.g., **N** and **B** as stated above. An empty set is denoted by $\emptyset$. We use the union symbol $\cup$ and the difference symbol $-$ to represent the insertion and deletion operations to a set, respectively. The power set of a set S is written as 2^S. If T is a sequence type, the projection operator can be applied to a set S of type $Set\langle T\rangle$, written $S{\downarrow}X$, producing the following set: $\{\, s{\downarrow}X \mid s \in S \,\}$.

The map data structure $Map\langle S, T\rangle$ is an associative container that maps unique keys of type S to values of type T. An empty map is denoted by $\{\}$. If m is a map, $m[x \mapsto y]$ represents the insertion or update of the key x with value y to m. The operator is naturally extended to $m[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$ to allow a simultaneous update of distinct keys $x_1, \ldots, x_n$. The value y of key x is represented by $m(x)$. If m does not have any value associated with x, then $m(x) = \mathsf{undef}$. Two maps m_1, m_2 of the same type $Map\langle S, T\rangle$ may be combined when for each key x of type S, either $m_1(x) = \mathsf{undef}$ or $m_2(x) = \mathsf{undef}$, or both. The combination of combinable maps m_1 and m_2, written $m_1 \cup m_2$, is m where for each key x, $m(x) = m_1(x)$ if $m_1(x) \neq \mathsf{undef}$, otherwise $m(x) = m_2(x)$. The predicate $diffOn(m_1, m_2, X)$ for a set of keys X is true if m_1 may differ from m_2 with respect to the values of keys in X and other keys are mapped to the same value.

[4] A sequence s is a *subsequence* of another sequence s' if s can be derived from s' by deleting some elements of s' while preserving the order of the remaining elements [Gus97, p. 4].

Part I.

Language and Component Framework

CHAPTER 2

Verification Framework Overview

The main aim of this thesis is to present a chain of verification methods from implementations to system models. This chapter provides an overview of how this chain works through a client-server example derived from an industrial Erlang case study by Arts and Dam [AD99].

Chapter outline. We first start by describing the case study in Section 2.1. Then, Section 2.2 discusses its implementation in αABS, which is described in Chapter 3. Finally, Sections 2.3 and 2.4 provide a sketch of the approach to verify the implementation against the requirements. This approach is formalized in Parts II and III which feature the client-server example as the running example.

2.1 Description

The case study deals with an implementation of a query mechanism of a distributed database. The database has a single front end which we call the server. The system consists of the server (sub)system and numerous clients. The server receives requests from clients, where each request contains a query. The server has to respond to the requests with the appropriate computation results. To serve each request, the server creates a worker and passes on the query to be computed. If a query can be divided into multiple chunks, more concurrent computations can be introduced in the following way. Before each worker processes the first chunk of the query, it creates another worker to which the rest of the query is passed on. When the computation of the first query chunk is finished, the worker waits for the result from the next worker and merges the computation results together. The merged result is then returned. From the client side, the server is expected to respond to each request with the correct computation result, without having to know how the computation is done. The structure that results from the response of the server to a request is portrayed in Figure 2.1 where a query can be divided into three chunks.

The requirements for the query mechanism highlights a number of challenges for the implementation. First, the number of clients and the number of requests

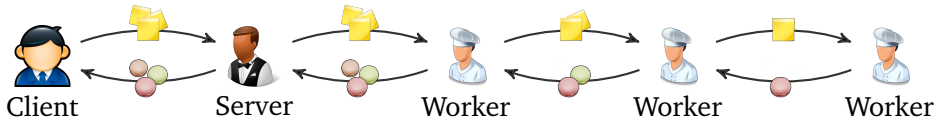


Figure 2.1.: Request processing structure on the server system[1]

being made to the server are unknown and can happen without any control from the server side. Second, there is no restriction on the size of a query. As a result, the number of workers needed to process a query is unbounded. The server and potentially each worker may need to coordinate the request with the computation result. In particular, the server must eventually respond each request with the correct computation result of the query if it is computed directly as a whole. The communication that happens between different requests must not interfere with or even block each other.

2.2 Implementation

We look into implementations of such distributed systems in terms of the actor model. The actor model is by nature open, so it is designed to handle situations where the system has no control over the environment it is running in. Furthermore, dynamic creation, such as the creation of the workers, is an inherent feature of the model. To coordinate the requests, a synchronization construct called *future* [BH77; Hal85; LS88] is added to the model.

As explained in Chapter 1, in the actor model, each entity of the system is represented by an actor. For our running example, each server and worker entity is represented by an actor. To describe the behavior of each actor, we choose an object-oriented approach, where the behavior of an actor is described by means of a *class* and each message sent to that actor has to correspond to a *method* of the class. The description is formalized through an object-oriented language called αABS, an actor subset of the Abstract Behavioral Specification (ABS) language [JHSSS11].

Listing 2.1 illustrates how the server scenario can be implemented in αABS. αABS has a Java-like syntax. The behavior of each server and worker actor is specified by its respective class. Each class implements an *interface* which defines the patterns of messages (i.e., *method calls* in object-oriented terms) an actor is allowed to receive from other actors. A standard static type system ensures that a class implementation adheres to these pattern restrictions, meaning that each actor only receives calls that are specified by the implemented interface. In other

[1] Icons in the figure are taken from `http://www.icons-land.com/`

Listing 2.1: Server implementation in αABS

```
interface IServer {
  Value serve(Query q);
}
interface IWorker {
  Value do(Query q);
}

class Server() implements IServer {
  Value serve(Query q) {
    IWorker w;
    Fut<Value> u;
    Value v;

    w = new Worker();
    u = w.do(q);
    await u?v;
    return v;
  }
}
class Worker() implements IWorker {
  Value do(Query q) {
    IWorker nxtWrkr;
    Fut<Value> u;
    Value v, c;

    if (querySize(q) > 1) {
      nxtWrkr = new Worker();
      u = nxtWork.do(restQuery(q));
      c = compute(firstQuery(q));
      await u?v;
      v = merge(c, v);
    }
    else {
      v = compute(firstQuery(q));
    }
    return v;
  }
}
```

words, an actor can only invoke a call on another actor, henceforth called the *target* actor, as long as that call matches the restriction of the target's interface. For example, the central class of the server scenario is `Server` which implements the interface `IServer`. All actors of class `Server` accept only `serve(q)` calls from other actors.

When a server actor receives a call `serve(q)`, it processes the query `q` and makes sure that the query is responded to. To enable concurrent computation of the queries, the server delegates each query to a dynamically created worker. If the query has more than one chunk (`querySize(q) > 1`), the worker delegates the rest of the query to a newly created worker and works on the first chunk. This delegation is done via a method call `do`, from which the server generates a *future*, a placeholder for the return value of the method call. This placeholder can only be filled in once, after which we say that the future is *resolved*. Its value can then be retrieved multiple times. The presence of futures allows the server to decouple the process of invoking a call from the process of retrieving the result. After working on the first chunk, the worker waits for the computation of the rest of the chunk to be finished by **`await`**ing for the future to be *resolved*. The **`await`** statement explicitly introduces a *processor release point*, which allows the server

to process other incoming requests instead of staying *idle* while waiting for the desired condition to be fulfilled. When the future is resolved, the returned value is fetched and `merge`d together with the computation result of the first chunk. The merged value is then returned. A similar process is also done by the server, but without the merging.

To complete the description, we assume appropriate definitions for the data types `Query` and `Value` and the total functions:

```
compute    :  Query ⟶ Value
querySize  :  Query ⟶ Int
firstQuery :  Query ⟶ Query
restQuery  :  Query ⟶ Query
merge      :  Value × Value ⟶ Value
```

where `compute(q)` computes the result of `q`; `querySize(q)` yields a number of chunks in which `q` could be partitioned; `firstQuery(q)` returns the first chunk of `q`; `restQuery(q)` returns the rest of `q`; and `merge` merges results. We also assume the following properties:

```
querySize(q) ≥ 1
querySize(q) > 1 → compute(q) = merge(compute(firstQuery(q)),
                                      compute(restQuery(q)))
querySize(q) = 1 → compute(q) = compute(firstQuery(q))
merge(null, v) = v
```

A query consists of at least one chunk; computing a non-primitive query is the same as merging the result of computing the first query with the computation of the rest of the query; computing a single query chunk is the same as computing the first query of the chunk; and merging `null` with some value `v` produces `v`. These assumptions follow the intention of the distributed query protocol [AD99], where the `merge` function acts as an aggregate to collect the partial subquery results from each database machine. Details on the precise semantics of the implementation appear in Chapter 3.

2.3 Specification

To discuss the verification of the server system implementation, we need to explore the requirements in more detail. Following Lamport [Lam83a] we first refine the requirements by means of operators. In terms of operators, the external requirement of the server system can be described as follows: every time the server receives a `serve(q)` call, its response is `compute(q)`. The link between these two operators is a future `u`.

Server system specification. A more formal way to write this requirement is by considering the desired behavior of the server system in terms of a set of traces of operators combined with a standard first-order logic. A trace is essentially a sequence of events generated by the execution.

$$\forall t, u, q : t \ \textbf{pr} \ldots \langle u \to s : \texttt{serve}(q) \rangle \ldots \langle u \leftarrow s : \texttt{compute}(q) \rangle \ldots$$

where s is the server actor and t is a trace of the server system. The formula above states that a desired trace t of the server system is such that for every `serve`(q) call directed to server s with future u attached to it, when the server finishes processing the query q, the resolved value of u is `compute`(q). The prefix operator **pr** allows t to represent an incomplete execution [BK08, p. 95], an execution that does not produce any more operators. While compact, the main problem with this formula is the precise meaning of the ellipses. If we replace the ellipses with any trace t_1, t_2, t_3, then we do not exclude the possibility that the `serve` operator with the same future u but with a different query q' may happen.

A workaround to this problem is by using projected traces. For example, we can project the trace t to the set of events that involves the future u, represented by $t{\downarrow}u$. The formula above becomes

$$\forall t, u, q : t{\downarrow}u \ \textbf{pr} \ \langle u \to s : \texttt{serve}(q) \rangle \cdot \langle u \leftarrow s : \texttt{compute}(q) \rangle \ . \tag{2.1}$$

In the formulation above, we do not accept traces that allow the presence of other `serve` operators (in fact any other operators) with the same future u.

Server actor specification. The projection operator works nicely as long as that the future resolution message is the only message that is being sent out. Consider now the requirement of the server actor. When the server actor receives a `serve(q)` call, it creates a worker and passes on the computation task to the worker. Whenever the computation is done, the server passes on the computation result back to the caller by resolving the associated future. A projection operator does not help us much here, because to specify this property, we have to provide a link between the request, the created worker, the future of the worker call. However, since we are on the class level, we have the guarantee that, for example, the worker creation is followed directly with passing on the computation task to the newly created worker. This observation allows a specification of this property as follows, assuming that we can fill the ellipses more precisely later on:

$$\forall t, u, q, w, u', v : t \ \textbf{pr} \ldots \langle u \to s : \texttt{serve}(q) \rangle \ldots \langle s \to w : \texttt{new Worker}() \rangle \cdot \\ \langle u' \to w : \texttt{do}(q) \rangle \ldots \langle u' \leftarrow w : \texttt{do} \triangleleft v \rangle \ldots \langle u \leftarrow s : \texttt{serve} \triangleleft v \rangle \ldots$$

The variables w, u', and v represent the name of the newly created worker actor, the future generated by the server actor to pass the request, and the value the worker has computed for the query q, respectively.

Excluding the ellipses, one problem with the specification technique above is that the represented traces do not allow the server actor to receive a call in between the creation of the worker and the `do(q)` call. Actors are inherently input-enabled as noted in Chapter 1. To solve this problem, Din et al. [DDJO12] introduce the *4-event semantics*, where they distinguish between the sending of a method-related operator and its reaction. For example, in this semantics the operator $\langle u \rightarrow s : \texttt{serve}(q)\rangle$ is split into two events:

- $\langle u \rightarrow s : \texttt{serve}(q)\rangle$, representing the sending of the method call, and
- $\langle u \twoheadrightarrow s : \texttt{serve}(q)\rangle$, denoting that s starts to react to the method call.

A similar split is also performed for the future resolution operator, hence the name 4-event. There are several reasons, which are elaborated in Chapter 4, why this splitting is useful for specifying the desired properties:

- The semantics provides the distinction between the events that are actually generated by the actor and the events that are not generated by the actor. Consequently, the specification can focus only on the events that are *generated* by the actor. The incoming messages can be uniformly treated by considering only *well-formed traces*, traces that adhere to the actor model characteristics.
- The semantics is more fine-grained than the operator model, allowing a more faithful representation of the asynchronous setting where there is a delay between the sending of a message, its receipt and the start of the reaction to the message.

Using the 4-event semantics, the previous specification can be localized to the server actor as follows.

$$\forall t, u, q, w, u', v : t \ \textbf{pr} \ldots \langle u \twoheadrightarrow s : \texttt{serve}(q)\rangle \cdot \langle s \rightarrow w : \texttt{new Worker}()\rangle \cdot \langle u' \rightarrow w : \texttt{do}(q)\rangle \ldots \langle u' \twoheadleftarrow w : \texttt{do} \triangleleft v\rangle \cdot \langle u \leftarrow s : \texttt{serve} \triangleleft v\rangle \ldots$$

The formula above can be grouped into two parts, focusing only to the events generated by the server actor. The first part states the reaction from the server actor until the introduction of the first release point where it waits for the computation result from the worker. The second part states the reaction from the server actor when the computation result is ready, that is, it resolves the future u. The return events are enriched with the method name, allowing for more information for the verification method to use. Together, these two parts can be seen as a *class invariant*, that describes the behavior of the server actor *between release points*.

We cannot use the projection operator to define the class invariant, because it throws away the information that there cannot be an event between, for example, the creation of the worker actor and the call to the worker. Instead, we add some state information as observed by Lamport when he wants to specify properties more precisely. Ideally the state should be as simple as possible to ease the verification. Building from the specification above, the state information should provide a relation between the future u of the `serve` call and the future of the worker u', as these information are needed in each part. The worker's name and the value returned by the worker can be inferred from well-formed traces. Taking the state to be a set of pairs $\langle u, u' \rangle$, the properties can be specified as transitions from a state to another, labeled with the desired sequence of events and enriched with proper preconditions.

$$\forall u, q, w, u', s : s \xrightarrow{\langle u \twoheadrightarrow s:\texttt{serve}(q)\rangle \cdot \langle s \rightarrow w:\texttt{new Worker}()\rangle \cdot \langle u' \rightarrow w:\texttt{do}(q)\rangle} s \cup \{\langle u, u' \rangle\}$$

$$\forall s, u, u', w, v : \langle u, u' \rangle \in s \implies s \xrightarrow{\langle u' \twoheadleftarrow w:\texttt{do} \triangleleft v\rangle \cdot \langle u \leftarrow s:\texttt{serve} \triangleleft v\rangle} s - \{\langle u, u' \rangle\}$$

The first formula states the change in the state after the server actor generates the three events in reaction to an incoming `serve` call, while the second formula states the change in the state when the server actor processes the return from the server. In the second formula, we have to ensure the proper coupling between the future u' the server generates in its `do` call to the worker and the future u of the `serve` call that causes the `do` call. By placing the existence of the pair $\langle u, u' \rangle$ in the state as a precondition, the second formula ensures that the transition occurs when the worker has finished its computation. Their conjunction becomes the class invariant for the server actor. The trace t is no longer part of the formula as it is represented by the state. Part II elaborates on the ingredients needed to complete a specification such as the class invariant above to produce an (actor) automaton semantics.

Server system specification revisited. Following the class invariant, the projection-based specification for the server system can also be transformed into a transition-based specification. For example, by setting the state to be a set of pairs $\langle u, q \rangle$ of the future of the incoming call and the corresponding query, we can rewrite Equation (2.1) as follows.

$$\begin{aligned} &\forall u, q : s \xrightarrow{\langle u \twoheadrightarrow s:\texttt{serve}(q)\rangle} s \cup \{\langle u, q \rangle\} \\ &\forall u, q : \langle u, q \rangle \in s \implies s \xrightarrow{\langle u \leftarrow s:\texttt{serve} \triangleleft \texttt{compute}(q)\rangle} s - \{\langle u, q \rangle\} \end{aligned} \tag{2.2}$$

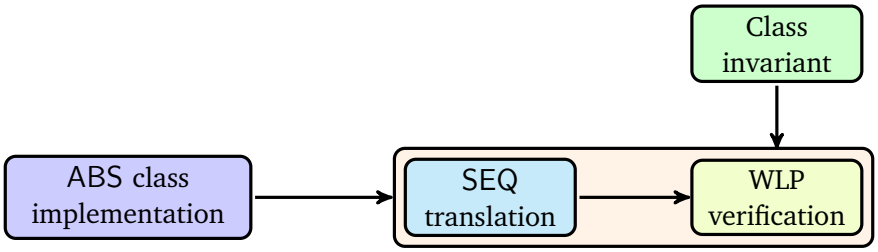


Figure 2.2.: Verifying class implementations

This transition-based specification is more readily translated to a (component) automaton and, thus, provides more support for the automaton model.

2.4 Verification

The specification technique outlined in the previous section is utilized for the two-tier verification approach. The first tier deals with the verification of the class implementations, while the goal of the second tier is to verify components/systems. As already mentioned in Chapter 1, we adopt the verification technique of Din et al. [DJO05; DDJO12; DDO12a] to handle the class verification. For the second tier, using the automaton semantics of the class and component specifications, the verification for components boils down to supplying a particular kind of simulation relation called a possibility map. In this section, we look into these two tiers in more detail using the server example.

2.4.1 Class Verification

The process we use for verifying the class implementation is illustrated in Figure 2.2. As input to the process for the first tier are the ABS class implementation and the desired class invariants. The process as proposed by Din et al. consists of two steps. The first step is the translation of a class implementation in ABS to a well-researched sequential language called SEQ [Apt81; Apt84]. One semantics of SEQ suited for verifying a class implementation against the invariants is the weakest liberal precondition (WLP) semantics [Dij75].

A program in a SEQ language consists of variable declarations, statements and procedures. A procedure is essentially the same as a method, except that it has no outer enclosure such as a class that a method has. A statement can be a standard sequential statement such as a variable assignment or a conditional branching, or a *non-deterministic* assignment, assume and assert statements. The non-

deterministic assignment is used to emulate various effects that cannot be (directly) controlled by an actor such as the return value of a method call and the name of a created actor. The assume and assert statements serve as the media to perform the invariant check. A WLP semantics of SEQ with these features is described by Apt [Apt84].

To have an idea how the resulting translation looks like, we translate fragments of the `Server` class into SEQ. The precise and complete translation appears in Chapter 11 (Listing 11.1). The main idea of the translation is to embed the trace information and the class invariant into the SEQ program, allowing a weakest liberal precondition based on them to be constructed. The translation keeps the statements whose constructs are present in SEQ as they are, such as the variable assignment and the conditional branching statements. Emulating actor creation, method call and future resolution statements that involve identity generation or return values is done by performing a combination of non-deterministic assignment, assume and assert statements. For example, the three statement sequence `w = new Worker(); u = w.do(q); await u?v;` in the `serve` method is encoded as follows.

```
// w = new Worker();
w' = some;
t = t·this → w' : new Worker();
w = w';
assume wf(t);

// u = w.do(q);
u' = some;
t = t·u' → w : do(q);
u = u';
assume wf(t);
```

```
//await u?v;
assert I(fields, t) && wf(t);
t' = some;
v' = some;
t = t·t'·u ↞ w : do◁v';
v = v';
assume I(fields, t) && wf(t);
```

In the translation above, the trace is stored by the variable `t`, and the check for trace well-formedness and class invariant are represented by functions **`wf`** and **`I`**, respectively. All three statements affect the trace such that each adds an event to the trace. The actor creation statement for example adds the creation event to the trace. Because we do not know the name of the created worker, a non-deterministic assignment is made to "generate" the name, such that when the event is added to the trace, the trace remains well-formed, as assumed by the **`assume`** `wf(t)` statement. The same translation method is applied for the method call translation, where the future identity is generated non-deterministically.

The **`await`** statement translation is more involved because of the introduction

of a release point. When a release point is made, other tasks may progress under the assumption that the class invariant holds. When the task is given the control back, the trace may have grown and the values of the fields may have changed. To emulate this correctly in SEQ, an assertion on the class invariant (and the well-formedness of the trace) is made before the release point. The class invariant is then assumed after progress on the other tasks is emulated by means of non-deterministic assignment. The non-determinism also includes the possibility that the task gets the control back immediately after the control is released.

After the translation is done, it is sufficient to check for each method whether given a well-formed trace and a class invariant implies the WLP of that method. This check is done by reasoning in first-order logic. When the implication holds for each method, the soundness of this verification method [DDJO12; DDO12a] allows us to conclude that the class implementation satisfies the class invariant.

The trace-based class invariants are the basis for specifying the actor automata, the basis of the DIOA model adaptation for actors. With a sound connection between class invariants and actor automata explained in Chapters 10 and 11, we can use the actor automata as the basis for verifying component/systems.

2.4.2 Component/System Verification

With the classes verified, we have the basis to verify if the whole system implementation satisfies its specification. The verification technique developed by Din et al. can be used to verify a system implementation by inferring the system specification from the invariants of each actor of the system. When these actors are fixed and the creation of actors does not depend on the system's interaction with its environment, this compositional approach is appropriate for proving that the system specification is fulfilled by the implementation. However, as seen in the server example, the creation of the worker actors depends on the query sent by the clients of the server. Therefore, the verification process must deal with an unbounded number of actors. To work around this issue, we introduce a *static* component notion.

The main idea is to designate a specific *activator class* and form a *component instance* based on the actors transitively created by an actor of that activator class. We call the initial actor of the component instance as the *head actor*. To illustrate this idea using the server system example: consider a server actor s that has received 2 queries. The first query consists of two subqueries while the second query consists of three subqueries. To process each query, s creates a worker actor that performs the computation. After completing the computation of all queries, the creation relation between the actors in the server system is captured by the

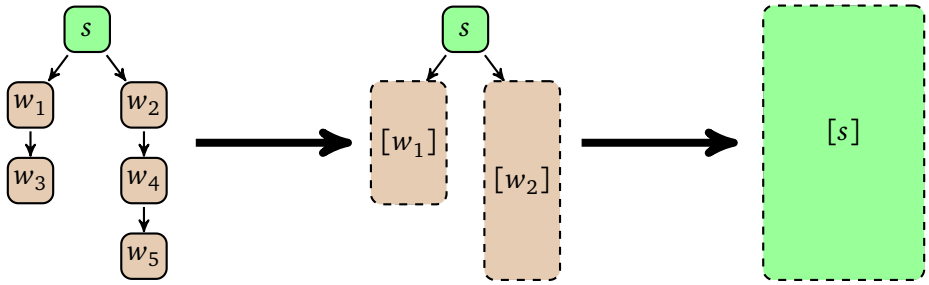


Figure 2.3.: Server system's actor creation structure after processing 2 queries and hierarchical componentization

left part of Figure 2.3. Because each worker actor has exactly the same behavior, we can represent the behavior of a chain of worker actors (e.g., the w_1, w_3 and w_2, w_4, w_5 chains) as a single instance. Taking the `Worker` class to be an activator class, these chains become component instances with w_1 and w_2 as the heads of these instances (middle part of Figure 2.3). With the same principle, we can also compose together these component instances with s and produce a component instance headed by s (right part of Figure 2.3).

What is common about these component instances is that each is derived from a specific class. The behavior of an instance of a class is predetermined by the class implementation. Consequently, the behavior of the component instance is also predetermined. The behavior of the server system can be represented by a *component* with the `Server` class as an activator class of that component. We have an entity, denoted by [`Server`], to refer to this component. This generic representation of such component instances allows a specification to target a part of the implementation, instead of some specific actor name of class `Server`. For example, the specification of the server system (Equation (2.2)) can be designated to [`Server`] instead of a server actor named s and the worker actors transitively created by s.

The verification on this tier is done by constructing a *possibility map* [LT87; NS94] from the *composed states* of the subcomponent specifications to the states of the component specification. Essentially, a possibility map requires the automaton representations of both subcomponents and the component to synchronize on external events, while allowing the subcomponents to progress when executing internal events. In its implementation, a component may dynamically create subcomponent instances. Therefore, a state of a component implementation may consist of several substates of the subcomponent instances. In the automaton model, this state is a Cartesian product of the substates of the subcomponent

specifications enriched with information on the created instances. If a possibility map can be constructed, the trace of an execution of the component implementation is a trace of the component specification. That is, the implementation does not produce some non-specified behavior.

For the implementation of the [`Server`] component, the external events are the `serve` calls and the returns, while the events between the server actors and the worker component instances are internal. A possibility map p for the [`Server`] component is roughly constructed as follows, where the states of the [`Server`] implementation $impl([\texttt{Server}])$ is a tuple $\langle ss, wn, m\rangle$ consisting of

- the server actor's state ss,
- a set of names of the [`Worker`] instance's head actors, and
- a map from the head actors to a [`Worker`] state.

$$\forall \langle ss, wn, m\rangle \in states(impl([\texttt{Server}])), y \in states([\texttt{Server}]) :$$
$$(First(ss) = First(y)) \implies p(\langle ss, wn, m\rangle) = y$$

The formula above maps a state of the [`Server`] component implementation to a state of the [`Server`] component specification when the set of futures stored in the server actor's specification state and the server component specification state is the same. The function *First* collects the first elements of all tuples in a set of tuples, which in the case of the server actor and the server component are the futures attached to the `serve` calls. It can be checked that the `serve` calls and returns are indeed simulated correctly when the component's implementation and specification are in the states where the mapping condition is fulfilled. The actual possibility map for the [`Server`] component and its implementation is more involved because the actor characteristics and the automaton representation must be taken into consideration. A more precise discussion on this possibility map is deferred to Chapter 12.

The rest of this thesis provides the details of this automaton-based verification framework, starting by providing a sound basis for the implementation language αABS.

CHAPTER 3

αABS: Syntax and Semantics

A sound basis for a system's implementation is crucial to perform formal verification. We base the implementation on the language αABS, a kernel object-oriented language that combines actor model with non-shared futures. This language can be seen as an intersection between the modeling languages Creol [JOY06] and ABS [JHSSS11] and an enrichment of Rebeca [SJ11].

Chapter outline. This chapter formally describes αABS and starts by explaining the features of αABS more thoroughly. Then, Section 3.2 describes the formal syntax of αABS and Section 3.3 presents its operational semantics. Apart from an operational semantics, we also describe a denotational semantics for the αABS, but this description is postponed to the next chapter where we have a better toolkit to express the denotational semantics. Finally, Section 3.4 discusses the features of αABS in the context of related work.

3.1 Features

αABS is a subset of ABS, a language developed for "design, analysis, and implementation of highly adaptable software systems"[1], where distributed, object-oriented systems are identified as the highly adaptable software systems [IT95]. To achieve this goal, ABS locates itself between design-oriented and architectural languages such as UML [RJB04], minimalistic and foundational languages such as π-calculus [Mil99], and object-oriented specification languages such as JML [LBR06]. This allows ABS programs (or more precisely, models) to be executable, while being more readily verifiable. ABS adopts the structures of full-scale programming languages such as Java and Scala, that permits a reliable translation of ABS programs to these languages [AØVWW13]. A generic algorithm, for example, can be written in ABS and verified, before translated to the full-scale languages for optimizations and actual usage. In fact, these aspects are exactly the advantages of using an abstract language [Pla13].

[1] HATS project description (`http://www.hats-project.eu/node/113`)

Following the actor model, communication between actors in αABS are performed only via asynchronous method calls. This means that a method call is *non-blocking*, allowing the code execution on an actor to directly proceed with the next statement. Any incoming call is transformed into a *task* whose duty is to execute the method body. An actor stores these tasks in its *buffer*.

Futures. A common pattern that appears in the actor model is the *request/reply pattern* [HO09]. The interaction documented in this pattern consists of two parts:

- an actor sends a request along with its name to another actor, and
- receives a reply message from that other actor when the request computation result is ready.

One approach to tackle this pattern without having to expose the actor's name is by using *futures* [BH77; Hal85; LS88]. Futures are proxies for values which are to be computed. A future is either *unresolved* or *resolved*, where resolving a future means that the future is populated with a value. Initially, a future is unresolved and once it is resolved, it cannot be resolved anymore. This resolution process is done transparently in αABS via method returns. To obtain the value of a resolved future, it has to be *claimed*, which is done *explicitly* in αABS. Apart from avoiding having to expose names, the use of futures also reduces the number of methods that need to be declared because the computation result can be retrieved without having to explicitly perform another method call. In αABS, the futures are *non-shared*, meaning that a future is accessible only by its generator and cannot be passed around as a parameter of a message. While less flexible than sharable futures, their introduction provides an interesting challenge in terms of task coordination and verification.

Cooperative multi-tasking. To coordinate the progress of a task with other tasks and the future resolutions, αABS adopts the *cooperative multi-tasking* approach [AGGS09]. In this approach, each task is categorized either as being *ready*, *suspended*, or *active*, such that a new task initially has a ready status. Following the actor model, there can be at most one active task at any time being executed by an actor. Other tasks can only be active if the active task *explicitly* gives up control or if it completes its computation.

In αABS, cooperative multi-tasking is presented through explicit introduction of *release points*. A release point is the point within an actor where an active task gives up control. A release point can be introduced because a task requires some condition on the internal state to be fulfilled or it waits for a future value to be resolved (i.e., the result of some method execution becomes available). It is also introduced by default when a task finishes its computation (i.e., it provides

a return value). When a release point is introduced, a ready task or an *enabled* suspended task is selected at random to become active.

Programming to interfaces. αABS follows the principle of *programming to interfaces* [GHJV95], where the declared types of actors are interface types. This principle ensures that communication between actors are done wholly through the given interfaces. In particular, each call that is sent to an actor can be processed by that actor. In the original actor model and in popular actor programming languages such as Erlang and Scala, this principle is not present, allowing actors to receive messages that may not be processed at all. Such a restriction is desirable for verifying open systems, as it limits the scope of communication whenever possible.

Further features are presented alongside the syntax description of αABS. Before proceeding, we remark that our presentation of αABS and its operational semantics follows the ABS language description [Häh+10; JHSSS11].

3.2 Syntax of αABS

This section provides the formal syntax of αABS. The syntax follows the Java syntax, except for aspects that deal with distribution (concurrency). The semantics of αABS follows closely the semantics of ABS that focuses on the asynchronous communication layer [Häh13]. Design decisions such as allowing actors to multitask can be inferred from the design decisions for ABS [JHSSS11] and is shortly discussed in Section 3.4. This language focuses on crucial factors affecting actor behaviors that we would like to capture by the automaton model: dynamic creation, dynamic topology and non-shared futures.

The syntax of αABS consists of two layers: the data type and "functional" layer and the object-oriented and distributed layer. The former provides a mechanism to represent abstract data structures and internal computations, while the latter deals with the description of actors and the communication mechanism between actors.

3.2.1 Data Type and Functional Layer

This layer defines the data types that can be used in an implementation and side-effect-free functions that manipulate these data types. In particular, it is used to represent internal data structures used by an actor without committing to a lower-level implementation. The language provides some basic data types such

T	$::= I \mid D \mid$ **Unit** $\mid$ **Int** $\mid$ **Bool** $\mid$ **String** $\mid \ldots$	types
Dd	$::=$ **data** $D[\langle\overline{T}\rangle] = [\overline{Cons}]$	data type declaration
$Cons$	$::= Co(\overline{T}) \mid$ **null**	constructors
F	$::=$ **def** $T\ fn([\overline{T\ x}]) = rhs$	function declaration
rhs	$::= e \mid$ **case** $e\ \{\ \overline{pat \Rightarrow rhs}\ \}$	function right hand side
pat	$::= \_ \mid val \mid e \mid Co(\overline{pat})$	pattern

Figure 3.1.: Syntax of αABS (1)

as the singleton type **Unit**, integer, boolean and string. The basic data types can be extended to allow user-defined data types and functions.

For example, using this layer the Query data type of the running example can be defined as follows.

```
data Query = SingleQuery(String) |
             CompoundQuery(String, Query);
```

The SingleQuery is a possible constructor for the Query data type and accepts one parameter of type **String**. Using the data type definition, the functions firstQuery and restQuery can be represented as follows, where the values are extracted by means of pattern matching.

```
def String firstQuery(Query q) = case q {
  SingleQuery(_) => q;
  CompoundQuery(sq, _) => SingleQuery(sq);
}
def String restQuery(Query q) = case q {
  SingleQuery(_) => q;
  CompoundQuery(_, rest) => rest;
}
```

The formal syntax for this part is given in Figure 3.1. It uses the following syntactic conventions: *italics* denote non-terminals in the grammar; a list of syntactic entities are denoted by an overbar, such as $\overline{e}$ for a list of expressions e; optional parts are surrounded by square brackets $[\ldots]$.

The set of types T consists of interface types I to be explained in the following subsection, data types D and the basic types. Data type declarations Dd specify the construction of user-defined data types D which may be parameterized with other types (in a similar manner to generics in Java). Each constructor has a name Co and a list of types acting as the parameters of the constructor. A function

TF	$::= T \mid \texttt{Fut}\langle T \rangle$	types with futures
P	$::= \overline{Dd}\ \overline{F}\ \overline{In}\ \overline{Cl}\ \{\ \overline{TF\ x};\ s\ \}$	programs
In	$::= \texttt{interface}\ I\ \{\ \overline{MS}\ \}$	interface definition
Cl	$::= \texttt{class}\ C(\overline{T\ p})\ \texttt{implements}\ I\ \{\ \overline{TF\ f := e};\ \overline{M}\ \{\ s\ \}\ \}$	class definition
MS	$::= T\ mtd(\overline{T\ x})$	method signature
M	$::= MS\ \{\ \overline{TF\ x := e};\ s;\ \texttt{return}\ e\ \}$	method definition
v	$::= x \mid f$	mutable variables
rv	$::= v \mid p$	variables
e	$::= \texttt{null} \mid \texttt{this} \mid rv \mid c \mid fn(\overline{e})$	pure expressions
s	$::= \texttt{skip} \mid s;s \mid \texttt{if}\ e\ s\ [\texttt{else}\ s] \mid v\ :=\ e \mid v\ :=\ \texttt{new}\ C(\overline{e})$	statements
	$\mid\ [v\ :=]\ rv.mtd(\overline{e}) \mid [v\ :=]\ v.\texttt{get} \mid \texttt{await}\ g$	
g	$::= e \mid v?v$	guards

Figure 3.2.: Syntax of αABS (2)

declaration F has a function name fn, a return type T and accepts a number of named parameters $\overline{T\ x}$. Using the parameters, the definition of a function can be an expression over the parameters or a nested case expression, allowing some expression e to be matched with a list of possible patterns. A pattern can be a wild card _ that matches everything[2], an expression or a constructor. The right hand side of the first matching pattern is evaluated as the result of the function. The next layer provides a more precise syntax of an expression.

3.2.2 Object-Oriented and Distributed (Concurrent) Layer

The second layer presents the imperative layer of αABS, where the cooperation between actors can be expressed through communication and synchronization. Following the same syntactic convention, the syntax for this layer of αABS is given in Figure 3.2. A program P consists a number of definitions of data types Dd, functions F, interfaces In, and classes Cl and a body that defines the initial activity (similar to the `main` method in Java).

An interface definition In states the publicly available methods of actors that provide the interface I. The interface names are used as types of actors. The methods are declared by means of their signatures MS. A method signature $T_{ret}\ mtd(\overline{T\ x})$ consists of a return type T_{ret}, a name mtd and formal parameters

[2]That is, only its existence is of interest.

$\overline{T\ x}$, where $T\ x$ denotes the variable name x and its type T. The return type and the types of the formal parameters may be interfaces or (basic) data types. We allow the variable names to be omitted from the signature when it is part of an interface definition.

A class definition *Cl* introduces the class name *C* and contains a number of fields and methods. It is equipped with a *class constructor* that allows an actor of the class to execute some statement *s* when it is created. Some of the fields are initialized by (an expression of) read-only parameters of the class when an actor is created. As syntactic sugar, other fields that do not have any initialization expression are initialized by their default values, e.g., by integer 0, `false` or `null` as in Java. Fields and class parameters are grouped together under the term *class attributes*. The types of fields can include future types `Fut`$\langle T \rangle$. Combined with the method signature, the typing ensures that futures cannot be shared among multiple actors. Each class implements an interface I. This means that each class must contain a method definition for each method signature that is present in I. For simplicity, we require that there is at most one method definition per method name *mtd* in the class declaration (i.e., no method overloading). An interface can be implemented by any number of class. The method bodies are implemented similarly to Java, except for statements related to concurrency.

A method implementation M contains a declaration of local variables and a statement. As with fields, some of the local variables are initialized by expressions on the method parameters. Other local variables are initialized by their default values. A statement can be a skip (`skip`), a sequential composition (`s;s`), a conditional (`if-else`), a variable assignment to the actor fields or the local variables (`var := expression`), a creation of new actors (`new C(`$\overline{p}$`)`), an asynchronous call (`a.mtd(`$\overline{p}$`)`), a return (`return value`), a conditional process release (`await guard`) and a blocking future resolution (`v.get`). For clarity (both here and in the semantics), the assignment operator is represented here by `:=` instead of just `=` which is used in the program examples. A skip statement is typically used as a valid representation of empty methods or constructors, which are syntactic sugar. A return statement may only appear at the end of the sequence. This return statement can be omitted when the return type of the method is `Unit`. A variable assignment can be combined with creation and call constructs (i.e., `var := new C(`$\overline{p}$`)` or `var := a.mtd(`$\overline{p}$`)`). The actor creation statement yields an actor name, whereas the call statement yields a future.

A release point is introduced when the conditional process release statement is executed. For the execution of the task to continue, the guard of the statement must be satisfied. There are two kinds of guards: a boolean expression e over the class attributes and a future resolution $v_u?v$. The boolean expression can be used

by the actor to synchronize between different tasks. The future resolution guard is satisfied when the future u represented by v_u is resolved. The value stored in the future u is assigned to v. When the actor needs to synchronize with the result of a method call, a blocking statement (v_u.**get**) can be used instead. No release point is introduced, and the actor continues its execution only when the future u is resolved.

For simplicity, a non-empty class constructor only contains method call and actor creation statements. This means that the class constructor performs no conditional checks, cannot explicitly block or introduce a release point, does no variable assignments apart from assigning variables to the generated futures and actor names. In other words, the constructor performs some initialization that cannot be performed through side-effect free expression evaluations.

An expression can be of the constant **null**, the self name reference of the actor **this**, a local variable or a method parameter x, a field f, a data constant c (including data type constructors $Co(\overline{e})$, strings and integers), or a function $fn(\overline{e})$. A function can be, among others, an arithmetic function on integers, string concatenation or a user-defined function as facilitated by the functional layer. An expression in our language is *pure*, meaning that its evaluation does not affect the current state of an actor. A reflective mechanism is not present, so the caller of a method call cannot be inferred from (the future attached to) the call.

3.3 Operational Semantics

The run-time semantics of αABS is given as a small-step, operational semantics under the assumption that we deal only with *type-correct* implementations. This issue is orthogonal to the focus of this thesis on verifying the dynamic behavior of the systems. A thorough treatment of the type system can be found in the ABS language description [Häh+10; JHSSS11].

The operational semantics is defined by *reduction rules* on *configurations*. The configurations contain the code being executed and the heap with the instantiated actors. A configuration is then represented by the parallel composition of these entities. This binary composition is associative and commutative, which allows us to focus on the interesting parts of the configurations in the reduction rules. A configuration is represented at run time as follows:

$$
\begin{array}{lcll}
K & ::= & a[C,\sigma,l] & \text{actor } a \\
 & | & u[a,\sigma,l,s] & \text{task with future } u \\
 & | & u[a,\sigma,l,v] & \text{completed task with future } u \\
 & | & K \parallel K & \text{composition}
\end{array}
$$

An actor $a[C, \sigma, l]$ has the name a, contains information about its class C, its instance state σ, and a lock to indicate whether the actor has an active task it is executing. σ is represented by a map from fields to values. The lock l either has the value of $\top$ and $\bot$ (i.e., of boolean type), indicating whether an actor is currently executing a task or not, respectively. The entity $u[a, \sigma, l, s]$ represents the task with future u of actor a representing the statement s the task needs to execute with a map from the local variables to values. The lock l indicates whether the task is active. The task is essentially the representation of an actor executing an asynchronous method call as we see in the reduction rules. The task name also corresponds to the future identity of that task. When the task is completed, the entity $u[a, \sigma, l, s]$ is substituted by another entity $u[a, \sigma, l, v]$ where v represents the resolved value for future u. The task is provided as a standalone entity instead of being part of the actor entity to simplify the reduction rules.

To specify the reduction rules, we add two constructs to the statement syntax of αABS given in Figure 3.2:

$$s ::= \ldots \mid \texttt{grab} \mid \texttt{release}$$

The statements **grab** and **release** handle the lock, allowing inactive tasks to acquire the lock of the actor. In particular, the execution of the **release** statement can be seen as the actual introduction of a release point in the actor. As we only have pure expressions, the order of how expressions are evaluated is not relevant. For simplicity, we use $\mathcal{E}(e)\sigma$ to evaluate an expression e with respect to some state σ.

The reduction rules of αABS follow a standard sequential programming semantics, except for the parts dealing with the concurrency constructs. In Figure 3.3, we provide a sample of reduction rules that deal with the cooperative multitasking approach. The reduction rules are of the form $K \rightsquigarrow K'$, reducing the configuration K to K'. They are performed under a fixed underlying program. See Appendix B for the complete operational semantics.

The two rules R-GRAB and R-RELEASE dealing with the **grab** and **release** statements, respectively, change the configurations according to their desired descriptions. The locks of the task and the actor are synchronized to ensure that at most one task is active. These two statements are introduced when a method is invoked. As described by rule R-CALL, a method call statement introduces the creation of a new task associated with the target actor. The statement this new task has to execute is the corresponding method body, padded with the **grab** and **release** statements. With the padding, the new task has to first obtain the lock and later release the lock after the execution of the method is finished. The generated future identity is stored in the assigned variable. We assume that an asynchronous

R-GRAB
$$a[C,\sigma,\bot] \parallel u[a,\sigma,\bot,\texttt{grab};s] \rightsquigarrow a[C,\sigma,\top] \parallel u[a,\sigma,\top,s]$$

R-RELEASE
$$a[C,\sigma,\top] \parallel u[a,\sigma,\top,\texttt{release};s] \rightsquigarrow a[C,\sigma,\bot] \parallel u[a,\sigma,\bot,s]$$

R-CALL
$$\frac{\begin{array}{ccc} u' \text{ fresh} & s'' = body(m(\overline{x}),C') & a' = \mathcal{E}(e')(\sigma' \cup \sigma) \\ \overline{val} = \mathcal{E}(\overline{e})(\sigma' \cup \sigma) & \sigma_{u'} = \sigma_{init}[\overline{x} \mapsto \overline{val}] & s' = \texttt{grab};repAwait(s'');\texttt{release} \end{array}}{\begin{array}{c} a'[C',\sigma'',l'] \parallel a[C,\sigma',l] \parallel u[a,\sigma,l,x := e'.m(\overline{e});s] \rightsquigarrow \\ a'[C',\sigma'',l'] \parallel u'[a',\sigma_{u'},\bot,s'] \parallel a[C,\sigma',l] \parallel u[a,\sigma[x \mapsto u'],l,s] \end{array}}$$

R-AWAIT
$$\frac{u' = \mathcal{E}(v)(\sigma' \cup \sigma)}{\begin{array}{c} u'[a',\sigma'',l',s'] \parallel a[C,\sigma',\top] \parallel u[a,\sigma,\top,\texttt{await } v?v';s] \rightsquigarrow \\ u'[a',\sigma'',l',s'] \parallel a[C,\sigma',\bot] \parallel u[a,\sigma,\bot,\texttt{release};\texttt{grab};\texttt{await } v?v';s] \end{array}}$$

R-GET
$$\frac{u' = \mathcal{E}(e)(\sigma' \cup \sigma)}{\begin{array}{c} a[C,\sigma',l] \parallel u[a,\sigma,l,x := e.\texttt{get};s] \parallel u'[a',\sigma',l',val] \rightsquigarrow \\ a[C,\sigma',l] \parallel u[a,\sigma[x \mapsto val],l,s] \parallel u'[a',\sigma',l',val] \end{array}}$$

R-RETURN
$$a[C,\sigma',\top] \parallel u[a,\sigma,\top,\texttt{return } e;\ \texttt{release}] \rightsquigarrow a[C,\sigma',\bot] \parallel u[a,\sigma,\bot,\mathcal{E}(e)(\sigma' \cup \sigma)]$$

Figure 3.3.: Reduction rules of αABS (selected statements)

call is always successful (i.e., the receiving actor is always able to create a task).

The **await** and **get** statements respectively provide the non-blocking and blocking alternatives to retrieving the return result of a method call. The rule R-AWAIT highlights that when the return result is not ready, a new release point is introduced. Other variants of the rule exist to handle the other cases. The rule R-GET ascertains that the reduction on the statement can only go ahead if the return result is ready. This rule is the only rule dealing with the **get** statement, so if a future is not yet resolved, the task cannot proceed with its execution and thus blocks.

A task terminates when the **return** statement is executed. Because this statement is placed at the end of a method body, we can process it together with the padded **release** statement (R-RETURN). The result of evaluating the expression e becomes the value held by the future u.

3.4 Discussion

Programming actor behavior. In general, programming and modeling languages that adopt the actor model facilitate the description of an actor's behavior in two ways: *update-based* and *class-based*. In the update-based approach, the behavior of an actor is written down in a procedure. This procedure contains a number of patterns that determine what messages are accepted by the actor. When an actor finishes reacting to an accepted message, the behavior of the actor is *updated* to a possibly different procedure. This approach is proposed in the original actor model [HBS73; Agh86] and is followed by languages such as Act1 [Lie87] and Erlang [Arm03; Arm10]. Because the description of the behavior of an actor dynamically changes, an actor needs to accept all messages and store them in its buffer, even if some of them may never be reacted to. While this provides a simple mechanism to define a fixed communication protocol, it complicates the verification effort, because we can send any kind of message to each actor.

In the class-based approach used by, e.g., ABCL [YBS86], SALSA [VA01], Rebeca [SJ11], Creol [JOY06], JCoBox [SPH10], and ABS [JHSSS11], each actor is assigned to a *class* since its creation. This assignment never changes throughout the lifetime of the actor. A class contains a fixed number of method definitions. The actor's behavior is modified through the execution of a method by changing the value of the internal variables. Further actions taken by the actor, including the reaction to *further* messages may be influenced by these changes. This static description of an actor's behavior is advantageous for verification, particularly in an open setting, because it limits the messages an actor can receive.

Guaranteed invariants. The operational semantics provides the guarantee that there can be at most one task that an actor is actively executing. This invariant is derived from the reduction rules that ensures that within two scheduling points, only one task of an actor has the lock. The semantics, however, does not guarantee that busy-waiting is void from the executions. For example, the rules R-AWAIT, R-RELEASE, R-GRAB are consecutively chosen to handle one task. The introduction of non-terminating executions is usually avoided on the implementation level of the language by using queues to fairly schedule the tasks (e.g., in JCoBox).

Futures and cooperative multi-tasking. There are many ways to introduce futures into a language. Using the classification provided by De Boer, Clarke and Johnsen [BCJ07], futures in αABS are non-transparent on the caller side and

non-shared. A *transparent* future means that the future cannot explicitly be handled in a program to write some value (on the target side) or to retrieve the value (on the caller side). Adopting a fully transparent future, such as done in Multilisp [Hal85] and ASP [CHS09], means that method calls tightly couple the caller and the target. In αABS, the transparency is only done on the target side, where the placeholder is populated only once (the same as promises [LS88], a variant of futures that ensures that the placeholder is populated once) when the `return` statement is executed. The caller side has more control on how to wait for the resolution of a future and how often the value can be retrieved.

The original actor model does not have the notion of futures. The addition of futures requires a control mechanism on the actors to retain the deterministic sequential execution property. Several languages such as Multilisp and ASP block the actor execution when an actor needs to fetch the value of a future that is not resolved yet. Creol, JCoBox, ABS, and αABS use the multi-tasking approach as described in this chapter. This approach allows for better usage of computing resource.

Futures that can be shared are futures whose identity can be passed as parameters of method calls. By having it non-shared as in Creol, the future can only be accessed by the actor that generates the future. While it is not as flexible as shared futures, the request/reply pattern as stated in Section 3.1 is solved without exposing the caller's name.

Class constructor. In αABS, a class constructor allows an actor to execute asynchronous calls and create new actors without having the trigger of a call from other actors. Such behavior is called *active* [JOY06]. Other than that, an actor typically *reacts* to calls from other actors.

Loop construct. As in Rebeca [SJ11], the syntax of αABS does not include a loop construct. This decision ensures that each time a method call is processed, only a *finite* amount of method calls and new actors are generated, as constrained by the original actor model [HBS73; Agh86]. A loop construct can explicitly be introduced as present in ABS. It also can be emulated in αABS by recursive asynchronous calls and explicit synchronization via `await` on a counter placed as a field. More precisely, other methods which are not involved in the loop are prefixed with an `await` statement on a condition that either the counter is inactive, meaning there is no loop being emulated, or the counter has not reached the desired value.

Access modifier. There is no need to have a `private`/`public` access level modifier as in Java, because the states of actors are fully encapsulated and all methods within the interface are public. All other methods are private.

Multiple object representation, inheritance and subtyping. As stated in the chapter introduction, αABS can be seen as an intersection between Creol and ABS. One of the intersections is with regards to the non-shared futures as discussed above. Another intersection is on how an actor is represented. In Creol, an actor is represented by one active object [LS95]. This is extended in ABS, where an actor is represented by a *cog*, concurrent object group [SPH10]. It allows an actor to have multiple service objects that interact with its environment, exposing different aspect of the actor. On a higher abstraction level [Häh13], the cog as a whole is an actor, because there can only be at most one thread of computation active at any time. With a richer state representation, an actor can simulate a cog. For example, an actor can store a list of objects it "contains" and a message is sent to the actor if the target object is in the list.

Another intersection is with respect to inheritance. In Creol, a class can inherit the implementation from multiple classes (also known as multiple inheritance [Kro85]), allowing code reuse in a similar way that is done in C++. Inheritance is not present in ABS. Code reuse is performed in ABS by *code deltas* [Sch10]. Code deltas allow parts of code to be added or deleted from an implementation based on some desired software configuration.

Both Creol and ABS have the same notion of subtyping, where an interface can be extended to include additional methods. These languages guarantee *type safety* that can be statically enforced. In other words, there will be no calls sent to an actor that is not within the interface that actor supports. Because αABS is simpler in this respect, we can reuse their type checking procedure and assume type safety when performing verification.

CHAPTER 4

Trace Foundation of Actor Systems

To form a solid foundation for verifying actor systems, we formally define the entities present in actor systems using mathematical constructs. The elementary entities in our setting are actors, messages, classes and futures. From these elementary entities, we build the notion of *events* which represent the communication that happen between the actors. In other words, events are what are *observable* from an actor when it is viewed as a black box.

Particularly interesting for verification is that events are the typical primary building blocks for defining a *fully abstract* semantics (see, e.g., [Bro02] in the context of communicating processes), a semantics which allows for equivalence behavioral check without additional information of the implementation [Plo77]. With a fully abstract semantics, the desired requirement of a system can be expressed using the same building blocks as those that are used to define the semantics of the implementations. With this in mind, it is useful to define a *denotational semantics* for αABS based on events.

Denotational semantics is a compositional semantics that associates a mathematical entity with a(fragment of a)n implementation. For αABS, the appropriate mathematical entity is a set of *traces*, i.e., sequences of events. A trace shows one possible interaction within an actor system and between the system and its environment. Taking a set of traces as the mathematical entity means that the semantics shows all possible interactions an actor system can have. This choice makes the denotational semantics as the bridge between the (class) specifications and the implementations (see Chapter 11).

Defining a denotational semantics for an actor-based language is non-trivial because of the presence of unbounded non-determinism [Cli81]. There are at least three factors that contribute to the non-determinism: the open nature of actors in accepting messages, the delivery process of messages, and the message selection process within actors. To cope with the non-determinism, we follow the *guess and merge* approach by Ahrendt and Dylla [AD12] where input events in the trace are guessed and only (sub)traces that lead to well-formed traces are merged together when composing different fragments of the language.

Chapter outline. The chapter begins with the formalization of events and traces (Section 4.1). Section 4.2 explains what the observable events for a group of actors are. Then, Section 4.3 presents a trace-based denotational semantics for αABS. The presentation and conventions used in this section are derived from the work by Ahrendt and Dylla [AD12]. We close this chapter with some discussion on different semantical models that are developed for the actor model (Section 4.4).

4.1 Actor Universe, Events and Traces

An actor system consists of a number of concurrently executing objects called *actors*. Each actor communicates with other actors purely by sending messages *asynchronously*. Each actor has a unique *name*, which is used to *target* where each message should be sent[1]. Each message may contain a number of parameters, some of which may be actor names. The actors whose names are contained within a message's parameters become *known* to the target actor when the target actor reacts to that message. These actors are also called *acquaintances*. An actor system is started by having some initial actors which may be passed on some initial messages. To conform with the open setting, we allow the entity that starts the system to be part of the environment which implies that the environment knows the initial actors.

To transfer the basic concepts and our language setting to the automaton model, we formalize the elementary building blocks: actors, messages and events. We also define functions to extract particular elements for the blocks. These functions are used exclusively for definitions related to the adaptation of the DIOA model, and are not used within the system implementation or specification.

To model actors with futures, we use the following universes (represented as sets):

- the universe of *actor(name)s* $a, b \in \mathbf{A}$,
- the universe of *future*(identitie)s $u \in \mathbf{U}$,
- the universe of *messages* $m \in \mathbf{M}$, and
- the universe of *classes* $C \in \mathbf{CL}$.

We also introduce the universe of *data values* $d \in \mathbf{D}$ to represent, for example, constant integer values, strings or, in our server example, queries. We say "actor a" to refer to an actor of name a. The name is unique, such that actors a and b are the same if and only if $a = b$. The behavior of each actor is represented by a class C. A class also determines what kind of messages an actor of that class can process, represented by $aMsg(C) \subseteq \mathbf{M}$. This function states which messages are

[1] This implies that the actor model does not provide an explicit support for message broadcasting.

allowed to be sent to the actor and which messages the actor can send to other actors. Note that this function may include messages, particularly the messages sent by the actor, that never appear in an actual execution. We overload this function with an extra parameter $type \in \{in, out, int\}$ to distinguish respectively which messages are part of the input interface of the class, which messages can be sent by the actor to another actor, and which messages an actor can send to itself, e.g., to trigger certain internal computations.

Futures are used to distinguish calls with the same method name and parameters, and play a role in the correct retrieval of the return value. As with actor names, they need to be unique. We conveniently choose futures $u \in \mathbf{U}$ to be structured such that we can retrieve their generator, denoted by $gen(u)$. We let the model of futures open and fix them according to specific needs as long as the uniqueness property is maintained.

A message m can either be an actor creation `new` $C(\overline{p})$, a method call $mtd(\overline{p})$ or a method return $mtd \triangleleft result$, where $C \in \mathbf{CL}$ is a class, mtd denotes some method name, $\overline{p}$ is a list of actual parameters and $result$ is a return value. A parameter or a return value can either be a data value $d \in \mathbf{D}$ or an actor name.

From this foundation, we build the set of events $\mathbf{E}$. Adapting the 4-event semantics proposed by Din et al. [DDJO12], we characterize an event as the *occurrence* of an actor *emitting* or *reacting* to a message. That is, an event $e \in \mathbf{E}$ represents the occurrence of a message $m = msg(e)$ being sent from the *caller* actor $a = caller(e)$ to the *target* actor $b = target(e)$ or being reacted to by b. If m is a creation message, b will be the name of the newly created actor while a is its creator. If m is a method call, a future $u = fut(e)$ is attached to the event. Since the caller information is contained within the future, the caller $caller(e)$ can be represented by $gen(fut(e))$. The function $param(e)$ extracts the parameters of the message $msg(e)$. The definition below formalizes the notion of events.

Definition 4.1 (Events):
Let $a, b \in \mathbf{A}$ be actor names, $C \in \mathbf{CL}$ a class, $\overline{p}$ a list of parameters, mtd a method name, and $u \in \mathbf{U}$ a future. An *event* $e \in \mathbf{E}$ is one of the following tuples:

Event status	Emittance	Reaction
Actor creation	$a \rightarrow b : \texttt{new}\ C(\overline{p})$	Not introduced
Method call	$u \rightarrow a : mtd(\overline{p})$	$u \twoheadrightarrow a : mtd(\overline{p})$
Method return	$u \leftarrow a : mtd \triangleleft v$	$u \twoheadleftarrow a : mtd \triangleleft v$

An event is formalized as a quadruple of an actor name or a future representing the caller, an actor name representing the target, the event *status*, where $\rightarrow$

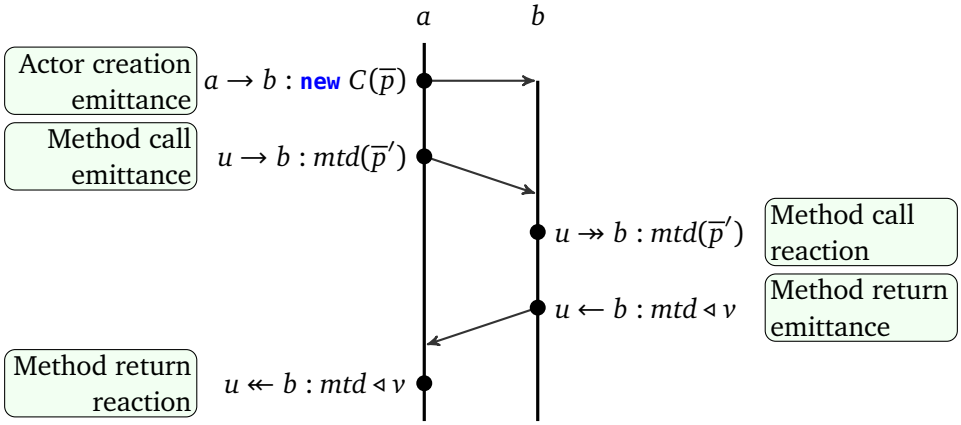


Figure 4.1.: Event types and their usage

and ← represents emittance and ↠ and ↞ represents reaction, and the message. The inclusion of the caller information, albeit implicitly through the futures for method calls and returns, significantly simplifies the presentation of the definitions of other mathematical entities. Following the assumption of successful message delivery mentioned in Chapter 3, an emittance event is carried out in synchrony with the receive of the event on the destination side. This means that the target actor stores the message in its buffer. A reaction event is an event when a message is actually processed by an actor. It can also be seen as a commitment made by the actor to process the message. A reaction event can happen only if the corresponding emittance event is present in the buffer of the actor. The asynchronicity of a method call or a method return is still represented by means of the buffer in each actor. The arbitrary delay between the emittance of a message and its reaction provides a more realistic view of the distributed setting, where sending a message takes time. For clarity, we sometimes notate an event e enveloped by angled brackets $\langle e \rangle$.

Actor creation is handled slightly differently from method calls. For simplicity we assume that actor creation is done instantaneously. Given an actor creation emittance event $e = a \rightarrow b : \texttt{new}\ C(\overline{p})$, the actor b is named such that $\mathit{creator}(b) = a$. The function $\mathit{created}(e)$ acts as an alias for $\mathit{target}(e)$ for creation events where it returns b. We also allow the function $\mathit{class}(e) = C$ to extract the class name C. Changes to include actor creation reaction events are relatively straightforward, but complicate the presentation of our automaton model.

Figure 4.1 illustrates what these events represent via two actors a and b and their execution lifelines (the parallel vertical lines). The black circles represent

the time when events are *generated* by the actor on whose lifeline the black circles appear. An outgoing edge from one lifeline to another lifeline indicates the source event is being sent from one actor to another. We use $Gen(a)$ to represent the set of events generated by a.

We can extract information from events in the following way. The predicates *isEmit* and *isReact* check whether an event is an emittance or a reaction event, respectively. If e is a reaction event, $emitOf(e)$ returns the corresponding emittance event. The predicates *isCreate*, *isCall* and *isRet* determine whether an event is an actor creation, a method call or a method return event, respectively. The predicate *isMethod* is used when the distinction between method call and method return is not necessary. As actor references are important to determine whom an actor can communicate with, we use the function $acq(e)$, short for acquaintance, to extract all actor names appearing in the message of the event. The information of the creator of an actor or the caller of a method call is transparent from the created actor or the target, respectively, so they are not part of the acquaintance.

We define the term *event core* as a triple of a future, a target actor and method name, written as (future, target actor, method name). For example, the event core of $u \rightarrow a : mtd(\overline{p})$ is $(u, a : mtd)$. We use the event cores to represent tasks an actor is currently handling, and thus they are not applicable to the creation events. The function *eCore* extracts the event core from an event.

Example 4.1.1 (Events):
The following provides several events that can be generated by an actor s of class `Server` from the client-server example.

- $e_1 = s \rightarrow w$: `new Worker()`
  The server s creates a new `Worker` actor w.
- $e_2 = u \twoheadrightarrow w$: `serve`(q)
  The worker w starts processing a request from some actor a that generates the future u to compute a query q.
- $e_3 = u \rightarrow w$: `do`(q)
  The server s sends a message to worker w to do the query q. The future u identifies this method call such that $gen(u) = s$.
- $e_4 = u \twoheadleftarrow w$: `do` $\triangleleft v$
  Given $gen(u) = s$, the server s fetches the resolved future u to obtain the value v.
- $e_5 = u \leftarrow s$: `serve` $\triangleleft v$
  The server s resolves the future u by filling it with value v.

Applying acq to e_1, e_2, e_3 and e_4 returns the set $\{w\}$, while for e_5 the result is $\{s\}$. The functions $creator(w)$ and $caller(e_1)$ return the actor s. △

To represent the interaction that happens between actors, we use the following simple notion of traces.

Definition 4.2 (Traces):
A *trace* $t \in Seq\langle \mathbf{E} \rangle$ is a sequence of events.

This simple definition allows us to lift up some functions that accept events as parameters to traces. For example, the function

$$created(t) = \{a \mid t' \cdot e \in Pref(t) \wedge isCreate(e) \wedge created(e) = a\}$$

returns a set of actors that are created in the trace t. This lifting also applies to the function $acq(t)$.

Because of its simplicity, it is not sufficient to restrict the kind of traces that may represent the interaction that happens. In particular, we need to take into account the characteristics of actors. For example, the underlying communication pattern used in the interaction needs to follow the pattern portrayed in Figure 4.1.

A trace that adheres to the actor characteristics is called a *well-formed trace*. To formally define this well-formedness properties, a trace is anchored to a certain set of actors A[2]. This anchor determines which relevant events need to be monitored for the well-formedness property to hold. We also use the following short hand notations for the projection operators. $t{\downarrow}(u, a)$ denotes the projection of a trace t to the set of events whose future is u and whose caller or target actor is a (i.e., $t{\downarrow}\{e \mid isMethod(e) \wedge fut(e) = u \wedge (caller(e) = a \vee target(e) = a)\}$). $t{\downarrow}a$ denotes the projection of t to the set of events where a is either the caller or the target actor (i.e., $t{\downarrow}\{e \mid caller(e) = a \vee (target(e) = a \wedge \neg isCreate(e))\}$). The creation event where a is created is not part of the projection.

The well-formedness of a trace is captured in three parts.

- The first part can be summed up as each future is used only for a unique communication cycle (with respect to an actor), as shown in Figure 4.1. There are three cases that needs to be handled: a call within the group, a method call from and to an actor outside of the group. The first case portrays the full communication cycle, while the latter two cases excludes reaction events generated from outside the group. Because the focus is on A, events between actors outside of A are ignored. We use the Kleene star for the method result reaction events because the value of a future may be fetched multiple times, and each event agrees on the fetched value.

[2]The anchor can also be the set of all actors **A** if we are dealing with a closed system.

Definition 4.3 (Well-formed traces):
Let t be a non-empty trace and $A \subseteq \mathbf{A}$ is a set of actors. The trace t is *well-formed* with respect to A if

1. the sequence of events related to a method call initiated by or targeted to an actor of the actor set follows the cycle of observable events depicted in Figure 4.1:

$$\begin{aligned}
&\forall u \in \mathbf{U}, a \in A : \exists b, mtd, \overline{p}, v : \\
&\quad (t{\downarrow}(u,a) \neq [\,] \implies u \rightarrow b.mtd(\overline{p}) \ \mathbf{pr}\ t{\downarrow}(u,a) \wedge (b = a \vee gen(u) = a)) \\
&\quad \wedge (gen(u) \in A \wedge b \in A \implies t{\downarrow}(u,a) \ \mathbf{pr}\ u \rightarrow b : mtd(\overline{p}) \cdot u \twoheadrightarrow b : mtd(\overline{p}) \cdot \\
&\qquad\qquad u \leftarrow b : mtd \triangleleft v \cdot [u \twoheadleftarrow b : mtd \triangleleft v]^*) \\
&\quad \wedge (gen(u) \notin A \wedge b \in A \implies t{\downarrow}(u,a) \ \mathbf{pr}\ u \rightarrow b : mtd(\overline{p}) \cdot u \twoheadrightarrow b : mtd(\overline{p}) \cdot \\
&\qquad\qquad u \leftarrow b : mtd \triangleleft v) \\
&\quad \wedge (gen(u) \in A \wedge b \notin A \implies t{\downarrow}(u,a) \ \mathbf{pr}\ u \rightarrow b : mtd(\overline{p}) \cdot u \leftarrow b : mtd \triangleleft v \cdot \\
&\qquad\qquad [u \twoheadleftarrow b : mtd \triangleleft v]^*) \ ;
\end{aligned}$$

2. the set of known actors grows monotonically:

$$\begin{aligned}
&\forall a \in A, t' \cdot e \in Pref(t{\downarrow}a) : isEmit(e) \wedge e \text{ is generated by } a \\
&\qquad\qquad \implies acq(e) \subseteq acq(t') \cup A \cup \{target(e) \mid isCreate(e)\} \ ;
\end{aligned}$$

3. each created actor has a fresh name:

$$\forall t' \cdot e \in Pref(t) : isCreate(e) \implies t'{\downarrow}target(e) = [\,] \wedge target(e) \notin acq(t') \ .$$

- The second part overapproximates the knowledge about other actors an actor has over time (cf. [AMST97]). As a consequence an actor a may never call another actor b whose name is still unknown to a. A trace does not provide the information when an actor forgets a name of another actor. The term e is generated by a refers, for example, to a method call emittance event where a is the caller or a method call reaction event where a is the target.

- The third part deals with the creation process. If a creates an actor, the created actor must be taken into consideration as part of the known actors, because the acquaintance function also includes the target actor of an event. That is, it locally ensures that no two actors will be created with the same name.

The next section confirms that the traces generated by an αABS implementation are well-formed.

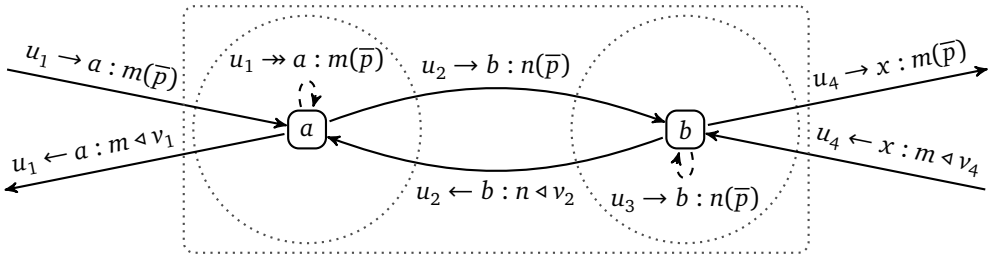


Figure 4.2.: Observable events of actors a and b and group of actors $\{a, b\}$

Example 4.1.2:
The following trace that is generated by the execution of two `serve` calls to server s is an example of a well-formed trace.

$u_1 \rightarrow s : \texttt{serve}(q_1) \cdot u_1 \twoheadrightarrow s : \texttt{serve}(q_1) \cdot s \rightarrow w_1 : \texttt{new Worker}() \cdot u_1' \rightarrow w_1 : \texttt{do}(q_1) \cdot$

$u_2 \rightarrow s : \texttt{serve}(q_2) \cdot u_2 \twoheadrightarrow s : \texttt{serve}(q_2) \cdot s \rightarrow w_2 : \texttt{new Worker}() \cdot u_2' \rightarrow w_2 : \texttt{do}(q_2) \cdot$

$u_2' \leftarrow w_2 : \texttt{do} \triangleleft v_2 \cdot u_2' \twoheadleftarrow w_2 : \texttt{do} \triangleleft v_2$

The first line considers the process of the first `serve` call until it has to wait for the computation result from the worker w_1. Similarly, the second is the trace s generates when processing the second `serve` call. The third line shows an incomplete response of the server when it receives the computation result for the second query. Note that the incoming call and method return events that are not generated by the server precede the reaction events.

The definition of well-formed traces allows for the second `serve` emittance event to be shifted anywhere in the trace prior to its reaction event. This flexibility reflects the input-enabledness of s. However, the method return emittance event for u_2' cannot be shifted any earlier, because that will violate the well-formedness condition. △

4.2 Observable Behavior

We consider events as the primary unit of *observability*. We say an event is *observable* by an actor if the actor receives or generates the event. This definition is naturally extended for a group of actors A to mean that an event is observable by A if there is an actor in A for which the event is observable. In Figure 4.2, all events are observable by the group $\{a, b\}$ with events with future u_1 are observable by a, events with future u_3 (which are an internal call and its not-shown reaction event) or u_4 generated by b are observable by b and events with future

u_2 generated by a is observable by both a and b. The notion of observable event of an actor can also be seen as all events that touches the execution lifeline (as portrayed in Figure 4.1) of that actor.

More interesting is what the *externally observable* behavior of an actor or a group of actors (for short, "entity") is. This information is relevant because it allows us to characterize how an entity behaves without needing to know how it works internally (i.e., a black box view). As we base the observable behavior on events, we need to define the notion of externally observable events. Intuitively, they are input and output events of the entity we are interested in. The entity develops a *boundary* which can be seen, e.g., graphically by the dotted borders in Figure 4.2. An externally observable event of an actor a is a non-self-call emittance event whose caller or target is a. An externally observable event of a group of actors A is an emittance event that either the caller is an actor of A and the target is not or vice versa. When a trace uses only the externally observable events of an entity, that trace is an *external trace* of that entity.

Example 4.2.1:
The following trace is an external trace of the actor a.

$$u_1 \rightarrow a : m(\overline{p}) \cdot u_2 \rightarrow b : n(\overline{p}) \cdot u_2 \leftarrow b : n \triangleleft v_2 \cdot u_1 \leftarrow a : m \triangleleft v_1$$

Note how the reaction events are not part of the external trace. △

The notion whether a creation event is externally observable depends on the context, whether the created actor resides within the boundary or outside. From an actor's perspective, the created actor is outside of the boundary and therefore the creation event is an externally observable event of the actor. If we consider a group of actors, a choice can be made to put the new actor within or outside the boundary of the group. For our purpose, we consider the new actor to reside within the boundary of the group. This choice coincides with the black box view we want to establish for components and systems as explained in the next chapter.

4.3 Denotational Semantics

The formal foundation explained in the previous section provides the building blocks to construct a trace-based denotational semantics for αABS. The denotational semantics of an αABS implementation produces a set of traces associated to the implementation. This trace set represents the overall behavior of the implementation paired with an unknown environment.

The basic idea for the construction of the denotational semantics follows the idea from Zwiers [Zwi89] where the set of traces of concurrent processes are built in two steps:

1. Construct independently all possible traces each concurrent process may produce, including *guessing* the observation the concurrent process may obtain from other processes.

2. Obtain the composed traces of the concurrent processes by *merging* traces of each concurrent process that *agree* on the same observations. Trace merging is defined as the inverse of a trace projection.

In terms of actors, a process is interpreted as a *thread* of computation that an actor makes when it receives a method call.

To obtain the desired semantics, Ahrendt and Dylla [AD12] push the guess and merge idea from Zwiers far. The guess part appears on several levels. First, in the semantics of a method, the parameters of a call are guessed. Then, the number of method calls (i.e., the task instances) for that method is also guessed as well as the number of actors instantiated from a call. All these guesses are used to populate the set of traces generated for one method execution, a method definition, and a class definition, respectively. To ensure that the interactions between these threads are appropriate, particularly that the executions of the sequential parts of method calls are uninterrupted between scheduling points, the semantics of an actor only includes mergeable traces. The merging conditions include an agreement over the states and a proper correspondence between emittance and reaction events.

The ingredients for defining the denotational semantics are the states of the actor and the traces. The states are denoted by σ, a map from variables to values. In the denotational semantics, the variables σ handles are the class attributes and local variables of method calls. We represent the maps of the class attributes by $\sigma|_{ca}$, while the maps of the local variables of the task k by $\sigma|_k$. These maps correspond to the σ in operational semantics configurations $a[C, \sigma, l]$ and $u[a, \sigma, l, s]$, respectively.

We also provide a model of tasks and futures suited for the denotational semantics. A task k is modeled by a triple $\langle a, mtd, i\rangle$ denoting the actor a it belongs to, the method it is executing and an integer. The integer serves as a unique identifier. A special task $k_{init} = \langle a, -, -\rangle$ is reserved for the actor's initialization process. When a constructor is present, this task executes the constructor. As the self reference of an actor is represented by the variable *this* in the state, the task identity is represented by the variable *me*. This variable is used only for the denotational

semantics and is not used within the implementation. A future u is a pair of tasks $\langle k_{caller}, k_{target} \rangle$ denoting the task that generates and processes the call, respectively.

The traces in the denotational semantics consist of events as described in the previous section. To allow the correct merging of the traces, information about when the scheduling points occur and the changes other tasks of the actor make (method call computations that execute while a task is suspended) needs to be present. This information is encoded by $yield(k, \sigma|_{ca})$ and $resume(k, \sigma|_{ca})$ that respectively represent when a task k becomes suspended and active. The parameter $\sigma|_{ca}$ allows a task to synchronize on the changes on the class attributes. We also append the creation events with the task k, written $k : a \rightarrow b : \texttt{new}\ C(\overline{val})$, to indicate which task performs the actor creation. Because the number of instantiations of a class is important, we let each actor a to contain its instantiation id, which is retrieved by the function $ciid(a)$ (short for class instantiation identifier). We denote the set of events extended by the extra information by **EE** and define traces for our denotational semantics as sequences of extended events $Seq\langle \mathbf{EE} \rangle$. The extra information is shed by means of projections to the set of events **E**, resulting in a set of traces.

The denotational semantics is represented by a function $[\![\,]\!]$ which comes in several flavors:

- a map from variable declarations, statements or method implementations and states to traces and states
- a map from method names, numbers of tasks and states to traces and states
- a map from method names and states to traces and states
- a map from classes and numbers of instances to traces
- a map from actors, classes and programs to traces

In this chapter we define $[\![\,]\!]$ for some selected statements, methods, classes and programs. See Appendix C for the complete denotational semantics.

Semantics for statements. A skip statement neither changes the state of the actor nor produces an event.

$$[\![\texttt{skip}]\!](\sigma) = \{(\sigma, [\,])\}$$

The sequential composition of statements records the changes and traces generated by the two statements in the sequential manner.

$$[\![s_1; s_2]\!](\sigma) = \{(\sigma_2, t_1 \cdot t_2) \mid \exists \sigma_1 : (\sigma_1, t_1) \in [\![s_1]\!](\sigma) \wedge (\sigma_2, t_2) \in [\![s_2]\!](\sigma_1)\}$$

The variable assignment statement updates the variable v in the state to the evaluation value of the expression e. If the evaluation always generates an error (e.g.,

division by zero), the resulting set is empty. This semantics is comparable to canceling the whole computation when an error happens as if the line of computation never happens, implying that the resulting trace sets do not include erroneous computations. This design decision allows for a cleaner composition of traces on method and actor levels (and also class and program levels in [AD12]). This approach is also followed every time an expression needs to be evaluated.

$$[\![v := e]\!](\sigma) = \left\{(\sigma', [\,]) \mid \exists val : val = \mathcal{E}(e)\sigma \wedge \sigma' = \sigma[v \mapsto val]\right\}$$

The actor creation statement adds the extended creation event to the trace. The actor identity of the newly created actor is guessed such that the actor creation relation and the class are appropriate.

$$[\![v := \texttt{new}\ C(\overline{e})]\!](\sigma) = \left\{ (\sigma', t) \;\middle|\; \begin{array}{l} \exists \overline{val}, a : \overline{val} = \mathcal{E}(\overline{e})\sigma \wedge \sigma' = \sigma[v \mapsto a] \wedge \\ \quad created(a) = \mathcal{E}(this)\sigma \wedge class(a) = C \wedge \\ \quad t = \mathcal{E}(me)\sigma : \mathcal{E}(this)\sigma \rightarrow a : \texttt{new}\ C(\overline{val}) \end{array} \right\}$$

The method call statement is treated by generating a future u, that guesses the task identifier on the target side. The method call emission event is the generated trace of this statement.

$$[\![v := v'.mtd(\overline{e})]\!](\sigma) = \left\{ (\sigma', t) \;\middle|\; \begin{array}{l} \exists a, \overline{val}, i, u : a = \mathcal{E}(v')\sigma \neq null \wedge \\ \quad u = \langle \mathcal{E}(me)\sigma, \langle a, mtd, i\rangle\rangle \wedge \sigma' = \sigma[v \mapsto u] \wedge \\ \quad \overline{val} = \mathcal{E}(\overline{e})\sigma \wedge t = u \rightarrow a : mtd(\overline{val}) \end{array} \right\}$$

The **await** statement is interpreted by means of the extra events *yield* and *resume* as the task releases the control to other tasks. The task only continues when the method return event is present in the buffer of the actor. The method reaction reaction event is appended to the trace. The resolved value of the future is stored in the state, to ease synchronization.

$$[\![\texttt{await}\ v?v']\!](\sigma) = \left\{ (\sigma', t) \;\middle|\; \begin{array}{l} \exists t', u, k_c, i, a, mtd, val : \\ \quad u = \mathcal{E}(v)\sigma = \langle k_c, \langle a, mtd, i\rangle\rangle \wedge \\ \quad (\mathcal{E}(u)\sigma \neq \mathsf{undef} \implies val = \mathcal{E}(u)\sigma) \wedge \\ \quad t = yield(u, \sigma|_{ca}[u \mapsto val]) \cdot \\ \qquad resume(u, \sigma'|_{ca}) \cdot u \twoheadleftarrow a : mtd \triangleleft val \wedge \\ \quad \sigma'|_k = \sigma|_k[v' \mapsto val] \end{array} \right\}$$

Semantics for methods. Given the semantics of the single statements, the semantics of a task is given below. As a denotation, we use the method definition as the representation for the task. A class constructor receives a similar treatment.

$$[\![mtd(\overline{x})\{\overline{TF\ y := e_{init}}; s; \textbf{return}\ e\}]\!](\sigma) =$$

$$\left\{ (\sigma_2, t_1 \cdot t \cdot t_2) \;\middle|\; \begin{array}{l} \exists \overline{val}, \sigma_1, u : (\sigma_1, [\,]) \in [\![\overline{TF\ y := e_{init}}]\!](\sigma[\overline{x} \mapsto \overline{val}]) \wedge \\ \quad (\sigma_2, t) \in [\![s]\!](\sigma_1) \wedge gen(u) = \mathcal{E}(caller)\sigma \wedge \\ \quad t_1 = resume(\mathcal{E}(me)\sigma, \sigma|_{ca}) \cdot u \twoheadrightarrow \mathcal{E}(this)\sigma : mtd(\overline{val}) \wedge \\ \quad t_2 = u \leftarrow \mathcal{E}(this)\sigma : mtd \triangleleft \mathcal{E}(e)\sigma_2 \cdot yield(\mathcal{E}(me)\sigma, \sigma_2|_{ca}) \end{array} \right\}$$

First, the state is updated with the parameter values and the initialization of local variables. This updated state is passed on as an input for the semantics of the method body. The trace generated by the execution of a single method call is essentially the trace generated by the method body. This trace is sandwiched by the method call reaction event and the method return emittance event. The information on acquiring and releasing control before and after the execution of the method call is then placed on the ends of the generated trace.

The semantics of a task provides the basis to define the composed semantics of a number of tasks i of the same method mtd. We use the function $mtdDef(C, mtd)$ to extract the method definition of method mtd from class C.

$$[\![mtd, i]\!](\sigma) = \left\{ t \;\middle|\; \begin{array}{l} t{\downarrow}\{\langle \mathcal{E}(this)\sigma, mtd, j\rangle \mid j \in \{1, \ldots, i\}\} = t \wedge \\ \forall j \in \{1, \ldots, i\} : \exists k, a, \overline{val}, \sigma_k, t_k, \sigma'_k : \\ \quad k = \langle a, mtd, j\rangle \wedge \sigma_k = \sigma[me \mapsto k, caller \mapsto a, \overline{f} \mapsto \overline{val}] \wedge \\ \quad (\sigma'_k, t_k) \in [\![mtdDef(class(\mathcal{E}(this)\sigma), mtd)]\!](\sigma_k) \wedge \\ \quad t{\downarrow}\{k\} = t_k \wedge cond_m(t) \end{array} \right\}$$

The semantics above demonstrates the general idea how to *merge* the different traces into one by means of *inverse of projection* [Zwi89]. The first line of the semantics ensures that the merged trace t of i different tasks does come from the traces generated by the tasks. We use the projection $t{\downarrow}K$ where K is a set of task identifiers as a shorthand for a projection to a set of events where tasks in K take part as the generator or the receiver of the events. Then, we attempt to create these i traces. First, we create the initial setting of each task, by populating the initial state with the task identifier, the reference to the caller and the field values. The initial values of the fields for each task may differ as it is influenced by other tasks that may have been previously executed. The result of executing the task under the given conditions is captured by (σ'_k, t_k). Then the merging of the traces of the tasks t_k is reflected as the inverse of projecting the merged trace t to each task. The predicate $cond_m(t)$ ensures that the switching of execution control between tasks only happen at release points, which is defined below.

$$\forall e_1, e_2, k_1, k_2, t_{k_1}, t_{k_2} : e_1 \cdot e_2 \subseteq t \wedge e_1 \subseteq t_{k_1} \wedge e_2 \subseteq t_{k_2} \wedge k_1 \neq k_2 \implies (e_1 = yield(\_) \wedge e_2 = resume(\_))$$

The release point is indicated by the consecutive *yield* and *resume* events in the trace. The underscore abstracts away from the irrelevant contents. The semantics of a method is the union over all possible numbers of tasks.

$$[\![mtd]\!](\sigma) = \bigcup_{i \in \mathbf{N}} [\![mtd, i]\!](\sigma)$$

Example 4.3.1:
To illustrate the denotational semantics of a method, consider the `serve` method of the `Server` class. The statement

```
w = new Worker(); u = w.do(q); await u?v; return v
```

generates a set of traces. Here is one of them, where there are two queries q_1 and q_2 that a server actor s receives, the task ids for each call are $k_1 = \langle s, \texttt{serve}, 1\rangle$ and $k_2 = \langle s, \texttt{serve}, 2\rangle$, and the corresponding futures of the calls are $u_1 = \langle k^1_{caller}, k_1\rangle$ and $u_2 = \langle k^2_{caller}, k_2\rangle$. The futures of the calls to the workers are guessed to be u'_1 and u'_2.

$$\begin{aligned}
&\mathit{resume}(k_1, \sigma_1|_{ca}) \cdot u_1 \twoheadrightarrow s : \texttt{serve}(q_1) \cdot \\
&\qquad k_1 : s \rightarrow w_1 : \texttt{new Worker}() \cdot u'_1 \rightarrow w_1 : \texttt{do}(q_1) \cdot \mathit{yield}(u_1, \sigma'_1|_{ca}) \cdot \\
&\mathit{resume}(k_2, \sigma_2|_{ca}) \cdot u_2 \twoheadrightarrow s : \texttt{serve}(q_2) \cdot \\
&\qquad k_2 : s \rightarrow w_2 : \texttt{new Worker}() \cdot u'_2 \rightarrow w_2 : \texttt{do}(q_2) \cdot \mathit{yield}(u_2, \sigma'_2|_{ca}) \cdot \\
&\mathit{resume}(u_1, \sigma''_1|_{ca}) \cdot u'_1 \twoheadleftarrow w_1 : \texttt{do} \triangleleft v_1 \cdot u_1 \leftarrow s : \texttt{serve} \triangleleft v_1 \cdot \mathit{yield}(k_1, \sigma'''_1|_{ca}) \cdot \\
&\mathit{resume}(u_2, \sigma''_2|_{ca}) \cdot u'_2 \twoheadleftarrow w_2 : \texttt{do} \triangleleft v_2 \cdot u_2 \leftarrow s : \texttt{serve} \triangleleft v_2 \cdot \mathit{yield}(k_2, \sigma'''_2|_{ca})
\end{aligned}$$

The trace above consists of 4 parts. The first and third parts concatenated together are the trace generated for processing the first call. The second and fourth parts (the shaded parts) represent the processing of the second call. The predicate $\mathit{cond}_m(t)$ allows the third part to be shifted around, but not before the first part because then the projection condition is not fulfilled. The traces generated by the denotational semantics has the all or nothing flavor (i.e., it describes only *complete* traces). A trace that reflects partial execution of a method (e.g., a trace with the fourth part only until the reaction to the method return event), is not part of the trace set.

The conditions on the denotational semantics so far already partially support the desired well-formedness property. For example, we can compare the trace above with the trace seen Example 4.1.2 without the input events for s. The rest of the denotational semantics definition strengthens the resulting traces so that the well-formedness property is achieved. △

Semantics for classes. The enumeration technique to obtain the semantics of a method applies as well for classes. Instead of the number of tasks, the enumeration is done on the number of instances of a particular class. First we consider the semantics of a single actor of class C, given the semantics of all methods in C. For simplicity we let $Mtd(C)$ to be the set of methods defined in C and assume there is no method name clash between classes. The class constructor, if present, is considered as a **`Unit`** method that does not produce any method return event.

$$[\![a]\!] = \left\{ t'' \;\middle|\; \begin{array}{l} \exists t, t' : t{\downarrow}Mtd(C) = t : \\ \quad (\forall mtd \in Mtd(C) : \exists t_{mtd}, \overline{val}, k, a'' : \\ \quad\quad t_{mtd} \in [\![mtd]\!](\sigma) \wedge t{\downarrow}\{mtd\} = t_{mtd}) \wedge \\ \quad t' = yield(\langle a, -, - \rangle, [this \mapsto a, \overline{ca} \mapsto \overline{val}]) \cdot t \wedge cond_a(t') \wedge \\ \quad (\forall a' : a' \neq a \wedge t'{\downarrow}CrEv(a, a') \neq [\,] \implies |t'{\downarrow}CrEv(a, a')| = 1 \wedge \\ \quad\quad \forall e \in CrEv(a, a'), e' : a' \in acq(e') \implies \neg(e' <_{t'} e)) \wedge \\ \quad (\forall e : isReact(e) \wedge e \subseteq t' \wedge emitOf(e) \subseteq t' \implies \\ \quad\quad emitOf(e) <_{t'} e) \wedge \\ \quad t'' \in addInput(k : a'' \rightarrow a : \texttt{new}\ C(\overline{val}) \cdot t', a) \wedge a = taskOf(k) \end{array} \right\}$$

The composed semantics of actors works in a similar way as for obtaining the semantics of tasks of the same method. The trace t is a merge of traces of the methods. In other words, the projection of t to the set of (extended) events that are generated from processing a method call *mtd* is a trace t_{mtd} that is part of the semantics of the method call. These events can be gathered via the method name in the task identifiers. We prefix this trace with a *yield* after initializing the class attributes and do some sanity check. First we check through the predicate $cond_a(t')$ that the class attributes of a have the same evaluation before and after a control switch happens.

$$\begin{array}{rl} cond_a(t') = & cond_m(t') \wedge \\ & \forall \sigma', \sigma'' : \exists k', k'' : yield(k', \sigma') \cdot resume(k'', \sigma'') \subseteq t' \implies \sigma' = \sigma'' \end{array}$$

Next we check that an actor a' created by a can only be created once as governed by the projection of the trace to $CrEv(a, a')$, a set of creation events where a creates a'. We enforce that a' can only be used after it is created. Then, any emittance event comes in the trace before the corresponding reaction event. (This occurs for self calls.) At this point, we also add the method call and method return emittance events generated by other actors through the function *addInput*. For every reaction event that appears in a trace of a whose emittance is not generated by a, the emittance event is inserted before the reaction event happens. A safeguard is present so that a method return cannot precede the call. This function also

guesses a method return emittance event, when the corresponding reaction event is not available (i.e., the actor does not retrieve the resolved value of a future). Furthermore, no events are added for internal events.

$$addInput(t', a) = \left\{ t \;\middle|\; \begin{array}{l} t{\downarrow}Gen(a) = t{\downarrow}Gen(a) \wedge \forall u : \exists e, e', t'' : \\ \quad (gen(u) = a \wedge target(u) \neq a \wedge t'{\downarrow}u = e' \cdot t'' \wedge \\ \quad\quad t'' = e^* \wedge eCore(e') = eCore(e) \wedge isRet(e') \implies \\ \quad\quad\quad t{\downarrow}u = e' \cdot emitOf(e) \cdot t'') \wedge \\ \quad (gen(u) \neq a \wedge target(u) = a \wedge t'{\downarrow}u = e \cdot t'' \implies \\ \quad\quad\quad t{\downarrow}u = emitOf(e) \cdot e \cdot t'') \end{array} \right\}$$

The addition of the emittance events connects the different actors when their traces are merged. This connection is visible through the set of events we pick as the common ground for merging. For an emittance event to appear on a trace of a receiver, this event has to really be generated, i.e., it appears on the trace of the generator. Only then the trace of the receiver and the trace of the generator can be merged together. In other words, the emittance events become the synchronization points between different tasks, and later on actors. This addition is artificial in the sense that the actual actor does not produce these emittance events, but it provides the connection between the automaton model and the class implementation (see Chapter 11).

Example 4.3.2:
The semantics for actors completes the trace described in Example 4.3.1 by providing the emittance events to the reaction events. For example, the `serve` call emittance events appear before their reaction events as can be seen below as one of the traces of the server actor s.

$$u_1 \rightarrow s : \texttt{serve}(q_1) \cdot resume(k_1, \sigma_1) \cdot u_1 \twoheadrightarrow s : \texttt{serve}(q_1) \cdot k_1 : s \rightarrow w_1 : \texttt{new Worker}() \cdot u_1' \rightarrow w_1 : \texttt{do}(q_1) \cdot yield(u_1, \sigma_2) \cdot$$

$$u_2 \rightarrow s : \texttt{serve}(q_2) \cdot resume(k_2, \sigma_2) \cdot u_2 \twoheadrightarrow s : \texttt{serve}(q_2) \cdot k_2 : s \rightarrow w_2 : \texttt{new Worker}() \cdot u_2' \rightarrow w_2 : \texttt{do}(q_2) \cdot yield(u_2, \sigma_1') \cdot$$

$$resume(u_1, \sigma_1') \cdot u_1' \leftarrow w_1 : \texttt{do} \triangleleft v_1 \cdot u_1' \twoheadleftarrow w_1 : \texttt{do} \triangleleft v_1 \cdot u_1 \leftarrow s : \texttt{serve} \triangleleft v_1 \cdot u_2' \leftarrow w_2 : \texttt{do} \triangleleft v_2 \cdot yield(k_1, \sigma_2') \cdot$$

$$resume(u_2, \sigma_2') \cdot u_2' \twoheadleftarrow w_2 : \texttt{do} \triangleleft v_2 \cdot u_2 \leftarrow s : \texttt{serve} \triangleleft v_2 \cdot yield(k_2, \sigma)$$

The condition $cond_a$ ensures that the changes on the state of s are preserved correctly at each release point.

The example trace above also shows that the insertion of the input events is done rather liberally. For example, the method return emittance event for u_2'

appears before the *yield* event of the first task. This flexibility does not pose a problem as the extra events are not explicitly used in the higher level semantics. △

Analogous to the semantics of methods, we first give the semantics of a class C having i instances (actors).

$$[\![C,i]\!] = \left\{ t \;\middle|\; \begin{array}{l} t{\downarrow}\{(C,j) \mid j \in \{1,\ldots,i\}\} = t \;\wedge \\ \forall j \in \{1,\ldots,i\} : \exists a, t_a \in [\![a]\!] : \\ \quad class(a) = C \wedge ciid(a) = j \wedge t{\downarrow}Gen(a) = t_a{\downarrow}Gen(a) \;\wedge \\ \quad \forall u : |t{\downarrow}\{e \mid isEmit(e) \wedge isCall(e) \wedge fut(e) = u\}| \leq 1 \;\wedge \\ \qquad |t{\downarrow}\{e \mid isEmit(e) \wedge isRet(e) \wedge fut(e) = u\}| \leq 1 \end{array} \right\}$$

This semantics is simpler for classes than for methods, because we are only dealing with message passing parallelism. We do not have to consider the shared state within a single actor. The first requirement for a trace t to be in the trace set is that each event e of the trace must involve at least one of the considered instances as an active participant. This condition is reflected by the projection $t{\downarrow}A$, which lifts $t{\downarrow}a$ from a single actor to the set of actors A. The second requirement states that the trace is merged from traces of the instances, such that the event ordering on the traces generated by an instance of C remains the same in the merged trace. Because the focus is only on the *generated* portion of the trace, we put in an additional condition on the method calls and returns, such that the added input events is present only once in the trace. This condition yields flexibility on the instances *not* to react to every input event coming their way, while maintaining the well-formedness of the resulting traces.

The semantics of class C is the aggregation of all possible number of actors of that class.

$$[\![C]\!] = \bigcup_{i \in \mathbf{N}} [\![C,i]\!]$$

Semantics for programs. Given a program P that has a set of classes $Cls(P)$, the trace semantics of P is defined as follows. We assume that the body of P is represented by a special class called `Main` containing a parameterless method *main*. The content of *main* is exactly the body.

$$[\![P]\!] = \left\{ t' \;\middle|\; \begin{array}{l} \texttt{Main} \notin Cls(P) \;\wedge \\ \forall C \in Cls(P), \exists t_C : t_C \in [\![C]\!] \wedge t{\downarrow}\{C\} = t_C \;\wedge \\ \exists t, a, u : t{\downarrow}Cls(P) = t \wedge class(a) = \texttt{Main} \wedge gen(u) = a \wedge t{\downarrow}u = [\,] \;\wedge \\ \quad t' = remCr(u \rightarrow a : main() \cdot t){\downarrow}\mathbf{E} \end{array} \right\}$$

Similar to the class semantics, the trace t is composed of traces t_C given by all the classes C of program P. Through the use of the projection operator $t{\downarrow}Cls(P)$, which is defined to be $t{\downarrow}\{a \mid class(a) \in Cls(P)\}$, we ensure that the trace t does come from the merging of traces in the class semantics. Then, the traces are stripped from the additional information needed to compose the traces of the individual elements of the method executions. This is done via the function *remCr* which removes the task information from an actor creation event, followed by a projection to the set of events **E** which throws away the *yield* and *resume* events.

$$remCr([\,]) = [\,]$$
$$remCr(e \cdot t) = \begin{cases} a \rightarrow a' : \texttt{new}\ C(\overline{val}) \cdot remCr(t) & \text{if } e = k : a \rightarrow a' : \texttt{new}\ C(\overline{val}) \\ e \cdot remCr(t) & \text{otherwise} \end{cases}$$

An important property that the denotational semantics has is that the resulting traces are well-formed. This property follows from the careful conditions placed on the definition of the semantics of each syntactic construct.

Lemma 4.1 (Well-formedness of the denotational semantics):
Let P be a program and t a trace in $[\![P]\!]$. Given a set of actors A where each actor is of class $C \in Cls(P)$, t is well-formed with respect to A.

Proof:
The first condition on the method call cycle follows from the definition of method calls, the definition of *addInput* and the sanity checks, particularly that any emittance event comes in the trace before the corresponding reaction event. The second condition on monotonicity of known actors follows from the predicates $cond_m$ and $cond_a$ that the knowledge of an actor is passed consistently from one release point to the next one and the merging restrictions. The third condition on the freshness of names of the created actors follows from the sanity check for actor creation events and the merging restriction on the definition of $[\![C, i]\!]$. □

From this lemma, we can derive the following corollary on the well-formedness of the traces of an actor. We use this result as the basis for proving a sound link between the verified class invariants and the automaton model (see Chapter 11).

Corollary 4.1 (Well-formedness of traces of an actor):
Let a be an actor and t a trace in $[\![a]\!]$. Then $remCr(t){\downarrow}\mathbf{E}$ is well-formed with respect to $\{a\}$.

4.4 Discussion

4-event semantics. Events are more commonly defined without any splitting [VT91; JO04; KBR05; DJO08; AD12]. That is, there is no distinction between the sending of a message and its receive or reaction. Instead of having two events[3] $a \rightarrow b : mtd(x)$ and $a \twoheadrightarrow b : mtd(x)$, there would only be one event: $a \rightarrow b : mtd(x)$. This means that both a and b deal with the same event, much in a similar vein to the merging between the traces.

At the first glance, this splitting introduces more complexity from the specification perspective as the alphabet is larger. However, it enables us to focus directly on what happens to a certain actor. If we look back at Figure 4.1, the 4-event semantics allows us to talk exactly what happens on one execution lifeline. Consequently, the model and the proof system (Chapter 11) becomes simpler as reasoning about a class can be done without considering the activity of actors in the environment [DDJO12].

Observable events of an actor. The denotational semantics of an actor $[\![a]\!]$ employs the set of input emittance events. As explained in Section 4.2, we consider these events as part of observable events of a. This view differs from Din et al. [DDJO12] where only the events generated by an actor are deemed observable from that actor. Indeed, the way we specify the automata (Chapter 10) supports this view. The benefit of including the input emittance events is that the automata can emulate the actions the environment makes. By showing a sound connection between the verification method of Din et al. and the automaton model for the classes (Chapter 11), we argue that this view difference is not significant.

Denotational semantics of Ahrendt and Dylla [AD12]. The denotational semantics of αABS is defined in a similar way as the denotational semantics of Creol proposed by Ahrendt and Dylla. Some of the definitions are tweaked to fit the context. Apart from syntactic differences of αABS and Creol, Ahrendt and Dylla use 2-event semantics instead of a 4-event semantics as the basis for the traces. As a result, they have to introduce more extra events to the trace set. In addition to the *yield* and *resume* events that appear in Section 4.3, a pair of extra events *invoc* and *comp* are needed to mark when a task execution starts and finishes, respectively. This extra pair of events are represented by the method call reaction and the method return emittance events.

[3] A slightly different notation where instead of futures, a represents the caller actor.

Ahrendt and Dylla leave many of the safe guards needed to ensure well-formed traces until the program level (i.e., $[\![P]\!]$). These safe guards essentially construct a partial order between the emittance and reaction events. We shift some of the burden to the actor level via the *addInput* function. Because of this choice, we can ensure that a trace of an actor is well-formed.

Other semantics for the actor model. Based on Agha's actor model [Agh86], Vasconcelos and Tokoro [VT91; Vas92] propose a Mazurkiewicz's trace model ([Maz86]). In contrast to the interleaving semantics used in this report, the traces are extracted from some concurrent semantics of an actor system which relies on an *independence* relation and the sequential behavior obtained from the implementation. This relation defines which pair of events that can occur simultaneously. Because the focus is on abstract description of the behavior of an actor system, the creation of actors are left out from the semantics. Extracting the independence relation from an implementation is challenging and how to compose the trace semantics of parts of a system is still an open question.

Talcott defines and refines several semantics for Agha's actor model in the open context [Tal98]. She compares the abstractness of each semantics and defines a notion of equivalence on each semantics to perform reasoning. Events used in these models are only of one kind: $\langle e : a \triangleleft m \rangle$, where e is a unique tag, m is a message (or method calls in our terms) and a is an actor to which m is targeted. The caller information is not present and the creation of actors are done implicitly. Futures are not featured.

The first model called *event paths* is an abstracted operational model based on rewriting logic [Tal96]. This model is based on *actor theory configurations* $(\rho, \chi)(B, E \mid h)$, each of which consists of an interface (ρ, χ) as explained above, a set of actors and their behavior B, a set of pending events E, and the history of the delivery of events h. The behavior of the actors is governed by rewrite rules which induce changes on an actor theory configuration. Each rule specifies what an actor may do when it consumes a message. More precisely, a rule application rewrites the configuration such that the event containing the consumed message is removed from E and appended to h, the behavior of the actor is updated, new actors along with their initial behavior are added to B, and the events generated by the actor is put into E. Two additional generic rewrite rules mimic the interaction with the environment: environment sending (input) messages to the system and the system delivering (output) messages to the environment. The rewrites produce labeled transitions, where the labels indicate the delivery of messages $\mathtt{d}(e)$, incoming input messages from the environment $\mathtt{in}(e)$, and outgoing output messages to the environment $\mathtt{out}(e)$. The sequences of transitions yield computa-

tions. An event path $(\rho, \chi)\overline{e}$ is the resulting interface and the sequence of labels of the transitions of a computation.

The second model is the *open event diagrams*, a generalization of event diagrams proposed by Grief [Gre75] and formalized by Clinger [Cli81]. An open event diagram $(\rho, \chi)\langle A, D, P, activator, \xrightarrow{arr}, acqB, Cr\rangle$ stores the information about the interface of a group of actors (ρ, χ), the set of internal actors A, the set of delivered events D, the set of pending events P, the *activator* function that associates to each event $e \in D \cup P$ the event that caused e to be generated, the arrival order $\xrightarrow{arr}$ of events to each internal actor of the group, the acquaintances *acqB* of each internal actor and the creation function *Cr* identifying the creator of each internal actor. Because an open event diagram does not provide a total ordering between the events, it may represent more than one event path. Similarly, an open event diagram may represent a number of traces of an ACA. However, it still represents a computation modulo permutation of the independent transitions (cf. the trace semantics by Vasconcelos and Tokoro), and does not constitute all possible behaviors of a component (instance).

The third model is the *interaction diagrams* which can be thought as a compact version of open event diagrams, focusing only on the external events. An interaction diagram $(\rho, \chi)\langle I, O, \prec, A\rangle$ keeps only an aggregated information in form of a partial order map $\prec$ from the input events in I to the output events in O. Because the ordering of the input events is not given, not every interaction diagrams is *admissible*. Only when such a well-formed total ordering called *global time* exists between the events is an interaction diagram admissible. Events in I correspond to the emittance input events, while events in O correspond to the emittance output events.

The fourth model is the extraction of *interaction paths* from interaction diagrams with respect to some global time. Formally, an interaction path is of the form $(\rho, \chi)\langle\overline{e}, A\rangle$, where the sequence of events $\overline{e}$ consists only of input and output events with the set of internal actors A needed for defining its composablility with another interaction path.

Compared to the denotational semantics described in Section 4.3, Talcott's models are more dynamic in nature, in the sense that the semantics include the actors that are initially present. The closest model to our denotational semantics is clearly the interaction paths. The main difference is that the sequence of events $\overline{e}$ an interaction path has consists only of the interaction of the group of actors with its environment. This representation fits to the notion of external traces described in Section 4.2. The idea of describing the externally observable behavior of a group of actors is explored in the next chapter, where we described the component notion.

Calculi for actors. Caromel et al. [CHS04; CHS09] propose a calculus, called ASP, to provide a minimal setting to study "object-oriented languages with asynchronous communications, futures, and sequential execution within each parallel process". This calculus extends the ς-calculus [AC96] by adopting the actors as objects and allowing the use of shared futures. Unlike the model discussed here, each actor can only handle one task at a time. Every time the value of a future is needed, the actor blocks until the future is resolved. A similar hierarchical structure of components is defined, where each component is represented by a fixed set of actors with fixed topology. The restrictions placed on the actors and the components are needed to ensure the system behaves deterministically. No explicit verification methods are given to check the correctness of the functional behavior of the system.

Gaspari and Zavattaro [GZ99] and Agha and Thati [AT04] develop process algebra formalisms of the actor model. The challenges with adopting process algebra such as CCS [Mil82], CSP [Hoa78] and π-calculus [Mil99] lie on the asynchronous communication and the persistent first-class actor names. Gaspari and Zavattaro follow the approach of asynchronous π-calculus [HT91] to model the asynchronous communication and the change of state on an actor is emulated by introducing a new name that is used only explicitly to execute the state change. However, their process algebra contain primitives that are not present in the actor model. Agha and Thati develop Aπ-calculus to get around these primitives by imposing type restrictions on π-calculus, similar in nature to what is done by Sangiorgi and Walker to model object-orientation in π-calculus [SW01, Chapter 10]. Neither of these calculi features the use of futures.

CHAPTER 5

Component Representation and Open Systems

Central to this thesis is the notion of components. Following Szyperski's definition of components [Szy98], the following requirements should be satisfied:

- Components are stateless.
- Components should allow the construction of increasingly large systems.
- Components need clear semantical interfaces that can hide internal behavior.
- Components can be independently deployable.

The stateless requirement hints that a component should be based on the *description* of the run-time entities. Only its *instances* have states. We can interpret the second requirement as finding a mechanism to reuse components to build bigger components. The interface requirement is identified as defining what a component *provides* to its users and what it *requires* to function properly. The last requirement can be roughly interpreted as a component being self-contained to enable its instantiation and encapsulate its features.

We identify these aspects in our setting by looking into the fundamental concepts. The actor model stipulates that a system is represented by configurations, each of which consists of actors [Agh86, Chapter 5]. Actors, however, are dynamic, run-time entities that have states. What is stateless is their description, which in Chapter 3 are classes. Classes are the building blocks for implementing a system. Moreover, a class has a clear interface that describes which methods an actor of that class provides and what the actor requires from other actors in order to operate. The provided interface of a class is described by the `interface` a class is implementing, while the required interface can be extracted from the implementation by analyzing the (parameters of the) method call and actor creation statements present in the implementation.

Using classes as the primitive entities, a component can roughly be seen as a set of classes. Consequently, when a component is instantiated, the instantiation is represented by a set of actors. Some questions still remain, namely

- what becomes the interface of a component,

- when a set of classes becomes independently deployable, and
- what the precise link of a component with the actors is.

Agha et al. [AMST97] answer the first question in the context of actors by keeping track which actors are exposed. Exposed actors are the medium for interaction, as an actor that is not exposed to any other actors cannot be a target of a call. The interface of a set of actors A has two parts $\langle \rho, \chi \rangle$:

- the (*receptionist*) actors ρ of A that are exposed to actors outside of A, and
- the (*external*) actors χ outside of A that are exposed to actors of A.

This interface may change as A interact with other actors, but the change is monotonic. That is, once an actor is a receptionist actor of A, it is always a receptionist actor (similarly for the external actors).

This answer is abstracted on the class level to some extent by analyzing the class implementation, particularly the class and method call parameters and the returns. Aggregating the interfaces of the classes whose actors can be exposed as a result of being passed around as parameters and returns produces an over-approximation of the provided and required interfaces of a component. A precise characterization of the interface of a component is still an active research field (see, e.g., [CNW13]).

The answer to the second and third questions depends on how a component is instantiated and what constitutes as the set of actors of such an instantiation. We combine two ideas: an *activator class* [Osg] and the creation relation between the actors [AMST97]. Designating an activator class to a set of classes means that when the component is instantiated, an actor a of the activator class is instantiated. This actor, called the *head* actor, becomes the initial member of the set of actors. Then, this set grows dynamically based on other actors a transitively creates (e.g., the worker actors transitively created by the server actor in Section 2.4.2). This set of actors is called a *component instance*, while the creation relation which forms a tree is called the *actor creation tree*.

To create an actor, the class of that actor must be known. Hence, this class must be part of the component. From this requirement, we can define a component to be independently deployable, if the class of any actor that can be created by the head actor is contained in the set of classes represented by the component. We name this kind of set of classes as a *creation-complete* set.

An effect of this requirement is that the behavior of all transitively created actors are known. However, it does not imply that the behavior of all actors that interact with the component instance is known. The behavior of external actors remains unknown. How the receptionist actors are used is also unknown. The distinction between known and unknown behavior allows for the categorization of actor systems into *open* and *closed* systems.

A closed system is essentially a system where it cannot influence and be influenced by the environment is running in. Otherwise, the system is open [HP85]. In terms of the interface of actor systems, a closed system is a system whose interface contains no receptionist and external actors [AMST97]. In other words, the key to a closed actor system is an empty interface: an interface that contains no method signatures. Because the head actor of a component instance is exposed to its creator, a component generally represents an open system. However, if the environment has no means to influence the behavior of the head actor after it is created the component represents a closed system. This is technically done by checking whether the activator class implements an empty interface.

Chapter outline. In this chapter we formalize the distinction between closed (Section 5.1) and open systems (Section 5.2). The chapter ends with some discussion on the component notion.

5.1 Closed Systems

To begin the exposition on closed and open systems, we first define what systems are. A system for a class-based actor language such as αABS is described by a set of classes. Within this set, there is a designated class, called the *activator class*, which is invoked to initialize the system, similar to the `Main` class concept in Java and many other class-based languages. The constructor of this class can be seen as the initial `main` method that is executed when an actor of this class is created. In αABS, a system can be represented by a program P by encapsulating the body of P in some fixed class which act as the activator class, as done for the denotational semantics $[\![P]\!]$ (Section 4.3).

Definition 5.1 (System):
A *system* is described by $Sys = \langle \mathbf{C}, C_0 \rangle$ is a set of classes $\mathbf{C} \subseteq \mathbf{CL}$ with a distinguished *activator class* $C_0 \in \mathbf{C}$.

One important characteristic of a class-based language is that to create an actor of class C, the corresponding class definition must be known. In terms of a system, C must be in the set of classes of the system. A system that satisfies this condition is called creation-complete.

Definition 5.2 (Creation-complete class sets):
Let $\mathbf{C} \subseteq \mathbf{CL}$ be a set of classes. $\mathbf{C}$ is *creation-complete* if for each actor creation message `new` C' in $aMsg(C)$ of any class $C \in \mathbf{C}$, $C' \in \mathbf{C}$.

Example 5.1.1:
The pair ⟨{`Server`}, `Server`⟩ is not a creation-complete system because a server may create a worker, yet the class `Worker` is not within the set of classes. The pair ⟨{`Worker`}, `Worker`⟩ on the other hand is a system because the only actors a worker may create are of the same class. △

A closed system is defined as a creation-complete system whose activator class implements an empty interface and has no class parameters[1].

Definition 5.3 (Closed systems):
A creation-complete system $Sys = \langle \mathbf{C}, C_0 \rangle$ is a *closed system* if the interface C_0 implements is an empty interface and C_0 has no class parameters.

Example 5.1.2:
Let `Main` be a class that implements an empty interface and whose constructor creates a server and calls the `serve` method. Then, the system ⟨{`Main`, `Server`, `Worker`}, `Main`⟩ is a closed system. △

When a closed system is executed, the environment cannot influence the execution of the instantiated system any longer because it cannot perform any method calls on the head actor. Furthermore, the instantiated system cannot influence the environment as no actors of the environment are exposed to the system to begin with. To show this property, we represent a closed system as a program P and show for every trace of the denotational semantics of P that there is no interaction between an actor of a system and an actor of the environment. To distinguish actors part of the system from actors part of the environment, we check whether the ancestors of an actor includes the head actor. We define the ancestors of an actor as a transitive closure of the actor creation relation.

$$ancestors(a) = \{a\} \cup \{a' \mid \exists a'' \in ancestors(a) : a' = created(a'')\}$$

Using this function, we over-approximate all actors that can be created by a.

$$descendants(a) = \{a' \mid a \in ancestors(a')\}$$

Lemma 5.1 (Closed system interactions):
Let program P be a closed system and a is the head actor. Then,

[1] The restriction on class parameters can be loosened up slightly by allowing data types that contain no actors to be part of the class parameters.

Lemma 5.1 (Continued)
$\forall t \in [\![P]\!], u : \exists u' : u' \rightarrow a : main()$ **pr** $t \wedge t{\downarrow}u \neq [\,] \implies$
$a \in ancestors(gen(u)) \wedge a \in ancestors(target((t{\downarrow}u)[1]))$.

Proof (by induction):
By Lemma 4.1 and Definition 5.2, a trace $t \in [\![P]\!]$ is well-formed with respect to $descendants(a)$. Therefore, the first element of $t{\downarrow}u$, when this projection does not yield an empty sequence, is a method call emittance event e. Let t', t'' be traces such that $t = t' \cdot e \cdot t''$. Then we show by induction on $t' \cdot e$ that the caller and the target of e are actors of the system.

- For the base case, e is a call to a invoking the method *main*. By the definition of $[\![P]\!]$, the caller is a. Thus, the property holds.
- For the inductive case, t' has no calls whose caller and target actors are part of the environment. Because `Main` implements an empty interface, a cannot be called by the environment to expose actors of the environment to a. Furthermore, `Main` has no class parameters. Consequently, $acq(t') \subseteq descendants(a)$. Because no other actor of the system is exposed to the environment, and no actors of the environment are exposed to the system, the caller and target of e must be actors of the system. Thus, the property holds. □

5.2 Open Systems and Components

Let us loosen up the constraint of the activator class, by allowing the environment to interact at least with the head actor. That is, we open up the system so it be influenced by the environment. This kind of systems is called open systems.

Definition 5.4 (Open systems):
An *open system* $Sys = \langle \mathbf{C}, C_0 \rangle$ is a creation-complete system with some activator class $C_0 \in \mathbf{C}$ such that C_0 implements a non-empty interface.

The relationship between an open system and a closed system is established by some *context*. The obligation of the context is to ensure the behavior of each actor is known. This obligation is fulfilled by plugging in some classes that are needed to close the system. The context may also use some classes that are used in the system.

Definition 5.5 (Context):
A *context* of an open system $Sys = \langle \mathbf{C}, C_0 \rangle$ is $X = \langle \mathbf{C}^x, C_0^x \rangle$ such that $\mathbf{C} \cup \mathbf{C}^x$ is creation-complete and $C_0^x \in \mathbf{C}^x$ implements an empty interface and has no class parameters.

The closure of open systems with appropriate contexts is described by the following proposition.

Proposition 5.1 (Open system + context = closed system):
Let $Sys = \langle \mathbf{C}, C_0 \rangle$ be an open system and $X = \langle \mathbf{C}^x, C_0^x \rangle$ a context of Sys. Then $Sys' = \langle \mathbf{C} \cup \mathbf{C}^x, C_0^x \rangle$ is a closed system.

Proof:
Follows from Definitions 5.3 to 5.5. □

The execution of an open system produces a trace where the environment can call methods of exposed actors of the instantiated system. More precisely, once an actor a of the instantiated system is exposed to the environment, then there is a context where a is called in any way providing it is consistent with the knowledge the environment currently knows about the exposed actors. To characterize this property, first we broaden the denotational semantics of αABS to cover systems by taking a similar semantics to that of programs.

$$[\![\mathbf{C}, C_0]\!] = \left\{ remCr(t)\downarrow\mathbf{E} \;\middle|\; \begin{array}{l} t\downarrow\mathbf{C} = t \wedge \forall C \in \mathbf{C}, \exists t_C : t_C \in [\![C]\!] \wedge t\downarrow\{C\} = t_C \wedge \\ \exists a, a', k, \overline{v} : \\ \quad class(a) = C_0 \wedge a \notin ancestors(a') \wedge \\ \quad t \neq [\,] \implies k : a' \rightarrow a : \texttt{new}\ C_0(\overline{v})\ \textbf{pr}\ t \wedge \\ \qquad\qquad\qquad\qquad created(t) \subseteq descendants(a) \end{array} \right\}$$

The denotational semantics of a system is a set of traces merged from the traces of classes, such that each trace begins with the creation of the head actor a and all created actors in the trace are descendants of a.

Important for the openness property is the exposed actors. Only through exposed actors can an environment interact with an open system. Given a set of actors A, some actor a is exposed to A if it is created by an actor of A or a is an acquaintance of an event directed to some actor in A.

Definition 5.6 (Exposed actors):
Given a set of actors A and a trace t, the actors exposed to A according to t are contained in the following set:

Definition 5.6 (Continued)

$$exposed(t, A) = created(t{\downarrow}\{e \mid isCreate(e) \wedge caller \in A \wedge created(e) \notin A\}) \cup acq(t{\downarrow}\{e \mid (isCall(e) \wedge target(e) \in A) \vee (isRet(e) \wedge caller(e) \in A)\})$$

In the previous section, we have seen how to split actors of a system and actors of the system's environment by referring to the descendants of the head actor. Based on this split, we can say an actor a of a system is exposed to the system's environment if the environment gains the knowledge of a through some prior interaction. Once a is exposed, we can expect that the environment may call a method of the provided interface (implemented by the class) of a at any time afterwards. It is not important which actor of the environment does the call, as the target actor does not get to access the caller information. Open systems fulfill this property as formalized by the following lemma.

Lemma 5.2 (Open system interactions):
Let a be the head actor of an instantiated open system $Sys = \langle \mathbf{C}, C_0 \rangle$, and the actors of the environment be represented by $A_{env} = \mathbf{A}{-}descendants(a)$. Then,

$$\begin{aligned}
&\forall t \in [\![\mathbf{C}, C_0]\!] : \exists a' \in A_{env}, \langle \mathbf{C}^x, \mathbf{C}_0^x \rangle, u : \\
&\quad a' \rightarrow a : \texttt{new}\ C_0(\overline{v})\ \textbf{pr}\ t\ \wedge \\
&\quad \langle \mathbf{C}^x, \mathbf{C}_0^x \rangle \text{ is a context of } Sys \wedge gen(u) \in A_{env} \wedge t{\downarrow}u = [\,]\ \wedge \\
&\quad \forall a'' \in exposed(t, A_{env}), m \in aMsg(class(a''), in) : \\
&\qquad isCall(m) \wedge acq(m) \subseteq exposed(t, A_{env}) \cup A_{env} \implies \\
&\qquad\quad \exists t' \in [\![\mathbf{C} \cup \mathbf{C}^x, C_0^x]\!] : t'{\downarrow}descendants(a) = t \cdot u \rightarrow m
\end{aligned}$$

The formula above represents the openness property by establishing an appropriate context that allows the generation of a trace of the system instance. This context is framed such that once an actor of the system is exposed to the environment, the trace can be extended by a method call to that actor. The trace of the system instance begins with the creation of the head actor by the environment. The provided interface of an exposed actor a'' is represented by the allowed input messages of the class of a''.

Proof (by construction):
Because of the interface model[2], we can assume without loss of generality that

- $\mathbf{C}^x \cap \mathbf{C} = \emptyset$, and

[2]The programming to interfaces principle avoids a problem called *replay* [Ste06]. The replay problem appears when the system instance exposes some actor to an actor of the environment whose class is part of the system. Because the implementation of the actor is fixed, that environment actor can only use the exposed component actor in a specific way. In particular, this environment actor may only store the exposed component actor and openness is not achieved.

- C_0^x is such that all necessary actors of the context are created before the activator class of the component is created.

Because the classes are disjoint, we can explicitly manipulate the behavior of each actor of the environment. The class $C \in \mathbf{C}^x$ has as many parameters as needed to receive the actors of the environment. For each method definition *mtd* of a class $C \in \mathbf{C}^x$, the acquaintance is stored in the state and disseminated to all other actors of the environment. Furthermore, *mtd* contains an internal call, whose method definition is recursively sending calls to the stored actors. By constructing such a context, given a trace t of the open system, the context can make it such that t is extended by a method call to some exposed actor. □

An important observation is that the set of classes of an open system can be constructed from the activator class. The proof of the following lemma shows how this is done.

Lemma 5.3:
Let $C_0 \in \mathbf{CL}$ be a class. Then there is a unique minimal set of classes $\mathbf{C} \subseteq \mathbf{CL}$ such that $\langle \mathbf{C}, C_0 \rangle$ is an open system.

Proof (by construction):
Let $consComp : 2^{\mathbf{CL}} \to 2^{\mathbf{CL}}$ be a function defined as follows

$$consComp(\mathbf{C}) = \mathbf{C} \cup \{C' \mid \texttt{new}\ C' \in aMsg(C) \wedge C \in \mathbf{C}\}$$

(i.e., a function that gathers all classes that can be created by classes in $\mathbf{C}$ including themselves). The subset relation forms a complete lattice and *consComp* is continuous, as we can only add new members in the set. Therefore, we can apply Kleene's fixed-point theorem and obtain a unique least fixed point of *consComp* applied to the activator class C_0. □

Components are generally open systems, as they commonly interact with their users. The lemma above allows us to separate components from open systems by eliminating classes that are not necessary for a component to function. Thus, we have a set of classes that are independently deployable without requiring the description of other classes.

Definition 5.7 (Components):
Let $Sys = \langle \mathbf{C}, C_0 \rangle$ be an open system. *Sys* is a *component* if there exists no subset $\mathbf{C}'$ of $\mathbf{C}$ such that $C_0 \in \mathbf{C}' \wedge \mathbf{C}' = consComp(\mathbf{C}')$.

Because the set of classes can be uniquely derived given an activator class, we can represent a component $Sys = \langle \mathbf{C}, C_0 \rangle$ simply by its activator class. We use the notation $[C_0]$, called the *boxed class*, to refer to component *Sys*. The classes in **C** are the *companion classes* of C_0. The universe of components is represented by $[\mathbf{CL}]$.

Example 5.2.1:
The pair $\langle \{\texttt{Worker}, \texttt{Server}\}, \texttt{Worker} \rangle$ is an open system, but it is not a component because a worker never creates a server. The pair $\langle \{\texttt{Worker}\}, \texttt{Worker} \rangle$ on the other hand is a component because the only actors a worker may create are of the same class. In addition, we can combine the `Worker` component with the `Server` class to create a new component: $\langle \{\texttt{Server}, \texttt{Worker}\}, \texttt{Server} \rangle$. These component pairs are referable by $[\texttt{Worker}]$ and $[\texttt{Server}]$. △

5.3 Discussion

The term *activator class* used in this chapter comes from OSGi [Osg], where their component is instantiated through `BundleActivator`. The model where the component is instantiated by instantiating a single actor of the activator class is not uncommon. For example, it coincides with the "actor adaptor" of CORBA Component Model [OMG06] and the "class factory" of COM [Mic99]. Should a need arise for having more than one initial actor, our model can simulate it by having the activator class as a stub that only creates the other actors.

An important part of a component notion is how the component is instantiated. When a component is instantiated, the resulting actors form a group which provides a clear *boundary* between one instance and the other. We identify three particular ways to group actors into instances of a component: static, programmer-defined and dynamic.

Static component instance. A static component instance contains all actors in **C**. As such, grouping the actors into the component is trivial to define, by following the class of each actor. However, the drawbacks of doing so are numerous. As the component instance contains all actors in **C**, every actor is at the boundary. This means that we cannot hide the internal behavior. Furthermore, the components then cannot share classes, as there is no way to separate the run-time instances of intersecting components. In our example, a static component instance of the $\langle \{\texttt{Server}, \texttt{Worker}\}, \texttt{Server} \rangle$ component includes all server and worker actors. Hence, we cannot focus only on a single server with the workers it creates to represent the run-time

view of the component. As a consequence, not only the focus on the activator class is lost, it is also difficult to specify the behavior that the component should have. Nevertheless, with this kind of component instances, we can apply abstraction techniques such as grouping together the buffers of different actors of the same class [DKOne]. D'Osualdo, Kochems and Ong utilize this abstraction technique for verifying safety properties of closed actor systems.

Programmer-defined component instance. A programmer-defined component instance contains all actors in the way how the programmer defines it by specifying at the point of creation to which component instance the newly created actor belongs to. For example, JCoBox [SPH10] does this by extending the `new` statement with `in` a to say that newly created actor resides in the same component instance as a. Various type-based ownership approaches can also be used (see, [CNW13] for state-of-the-art). This approach is the most flexible as it provides fine-grained information which actors are at the boundary. However, this leads to additional constructs which makes it more complex to handle.

Dynamic component instance. A dynamic component instance contains all actors that are created directly or indirectly by the initial actor of the activator class. In other words, the component instance is formed from the actor creation tree, with the initial actor of the activator class as the head. This gives a more fine-grained grouping than the static approach, but less specific than the programmer-defined approach. Additionally, we can keep track of which actors are on the boundary, while keeping track which new actors are included in the component instance. This focuses the attention on the behavior at the component's boundary. After all, the hidden actors do not appear at the boundary and hence hiding them reduces the communication with the context. Because of its dynamicity, it is less straight forward than the static approach to provide upfront the exact instances of a component. In this thesis, we follow the dynamic approach of identifying component instances which allows the use the activator class to represent the component. We generalize the setting given in [KPH13] to deal with non-shared futures.

Part II.

Automaton Framework for Actor Systems

CHAPTER 6

Dynamic I/O Automaton Model

As described in Chapters 1 and 2, dynamic creation and dynamic topology are prevalent features of actor systems. However, automaton models tend not to provide direct support for dynamic creation and dynamic topology. A promising proposal that supports dynamic creation is the dynamic I/O automaton (DIOA) model [AL01; AL15].

The basis of the DIOA model is the I/O automaton model [LT87], which is designed especially to represent open systems [FL05]. As typical in other automaton models, an I/O automaton consists of states and labeled transitions. What is distinct about I/O automata is the *categorization* of the labels of the transition, called *actions*, into *input*, *output* and *internal*. Input and output actions represent the communication between the entity represented by the automaton and its environment. Together these actions are called *external* actions. An internal action is an action that is visible only to the entity itself. Collectively, the actions and their categorization are called the *signature* of the automaton.

An action is called *enabled* on a certain state if there is a transition from that state with that action as the label. The I/O automaton model requires that each automaton is *input-enabled*, meaning that regardless the state, each input action in the signature must be enabled. This requirement nicely reflects the open system setting and is beneficial to detect serious errors when components of a system face unexpected inputs [Lyn96, pp. 202–203].

To support dynamic creation, the DIOA model extends I/O automata in the following way:

- Instead of having a generic one automaton model, the dynamic I/O automaton model introduces two layers of automata (Figure 6.1): the signature I/O automata (SIOA) and the configuration automata (CA). The SIOA represent the behavior of the entities on their own, while the CA represent the collective behavior of these SIOA while keeping track which entities are present (i.e., alive). A CA itself is actually an SIOA *derived* from the SIOA that represent the entities, such that each state of the derived SIOA is mapped to a *configuration* that describes the present SIOA and the current state of each of those SIOA.

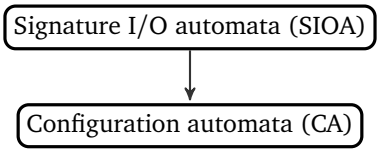


Figure 6.1.: DIOA model

- The SIOA extend I/O automata by allowing the signatures to change depending on the states. This notion of *state signature* is motivated by the dynamic creation aspect, which may cause new actions to be introduced into the signature and present actions to be eliminated or recategorized.

Allowing signatures to change on SIOA, particularly the input signatures, means that certain assumptions about the environment can be encoded. Being more explicit towards what the environment can do means that the verification effort does not have to deal with impossible actions by the environment.
Chapter outline. The description of the DIOA model is presented in two sections. Section 6.1 details the SIOA, while Section 6.2 defines the CA. The notation and convention used to describe the DIOA model follows from Attie and Lynch's latest technical report [AL15]. Section 6.3 provides a short discussion on the differences of the DIOA model presented in this chapter to that of Attie and Lynch. Discussion on other models suited to represent actor systems is postponed to the following chapters, where the discussion is more appropriate.

6.1 Signature I/O Automata

The general structure of an SIOA is captured by a *signature automaton* (SA): a transition system with functions that map each state of the automaton to the sets of input, output and internal actions that the automaton may take in that state.

Important to an SA is its *state signature*: a description of its input, output and internal actions parameterized by the state. Assuming a universal set of *actions* **Act**[1], a state signature $sig(\mathcal{A})(s)$ of an SA $\mathcal{A}$ in state s is a triple $\langle in(\mathcal{A})(s), out(\mathcal{A})(s), int(\mathcal{A})(s)\rangle$ representing the input, output and internal actions, respectively. The external signature of $\mathcal{A}$ in state s is defined by $ext(\mathcal{A})(s) = \langle in(\mathcal{A})(s), out(\mathcal{A})(s)\rangle$. Given a signature, the $\widehat{\ }$ operator yields the union of sets of the signature tuple, e.g., $\widehat{sig}(\mathcal{A})(s) = in(\mathcal{A})(s) \cup out(\mathcal{A})(s) \cup int(\mathcal{A})(s)$. The set of actions $\mathcal{A}$ could execute is represented by $acts(\mathcal{A}) = \bigcup_{s \in states(\mathcal{A})} \widehat{sig}(\mathcal{A})(s)$. We let action l (from *l*abel) to be a typical element of $acts(\mathcal{A})$.

[1]For the actor model, the universe of actions is represented by the universe of events **E**

Definition 6.1 (Signature Automata):
A *signature automaton* $\mathcal{A} = \langle states(\mathcal{A}), start(\mathcal{A}), sig(\mathcal{A}), steps(\mathcal{A})\rangle$ is a 4-tuple where

- $states(\mathcal{A})$ is a set of states,
- $start(\mathcal{A}) \subseteq states(\mathcal{A})$ is a non-empty set of initial states,
- $sig(\mathcal{A})$ is a signature mapping where for each $s \in states(\mathcal{A})$, $sig(\mathcal{A})(s) = \langle in(\mathcal{A})(s), out(\mathcal{A})(s), int(\mathcal{A})(s)\rangle$ and $in(\mathcal{A})(s), out(\mathcal{A})(s), int(\mathcal{A})(s)$ are sets of actions.
- $steps(\mathcal{A}) \subseteq states(\mathcal{A}) \times acts(\mathcal{A}) \times states(\mathcal{A})$ is a transition relation.

We assume the set *Autids* of SA identifiers, a universal set of signature automata **SA** and a map $aut \in Map\langle Autids, \mathbf{SA}\rangle$ such that $aut(\mathcal{A})$ is the SA with identifier $\mathcal{A}$. These identifiers uniquely represent the signature automata, in the sense that two different identifiers always refer to two different SA. We write "the SA $\mathcal{A}$" to say "the SA with identifier $\mathcal{A}$". The identifiers are used in the definition of CA to determine whether an SA can be inserted into a configuration.

We write $s \xrightarrow{l}_{\mathcal{A}} s'$ to represent a transition $(s, l, s') \in steps(\mathcal{A})$. The state s is called the *pre-state* of the transition, where as s' is called the *post-state*. We drop $\mathcal{A}$ from the notation when it is clear from the context. Following the I/O automaton model, typical constraints of I/O automata are placed on signature automata.

Definition 6.2 (Signature I/O automata [AL15]):
A signature automaton $\mathcal{A} = \langle states(\mathcal{A}), start(\mathcal{A}), sig(\mathcal{A}), steps(\mathcal{A})\rangle$ is a *signature I/O automaton* if

1. $\forall (s, l, s') \in steps(\mathcal{A}) : l \in \widehat{sig}(\mathcal{A})(s)$.
2. $\forall s \in states(\mathcal{A}) : \forall l \in in(\mathcal{A})(s) : \exists s' \in states(\mathcal{A}) : s \xrightarrow{l} s'$.
3. $\forall s \in states(\mathcal{A}) : in(\mathcal{A})(s) \cap out(\mathcal{A})(s) = in(\mathcal{A})(s) \cap int(\mathcal{A})(s) = out(\mathcal{A})(s) \cap int(\mathcal{A})(s) = \emptyset$.

The definition above states when an SA is an SIOA. The first constraint ensures that only actions that belong to the state signature may be executed in a transition. The second constraint ensures that the SIOA is input-enabled. The third constraint requires that the elements of the state signature are always pairwise disjoint. Note that there is no necessity for an action to remain as an input (or output or internal) action in all states of the SIOA. Of particular interest is that we can expand the

actions allowed in the signatures only as we need them (e.g., actions that only occur after some SIOA is created).

As with general transition system (cf. [BK08, p. 24]), the behavior of an SIOA is formalized using the notion of executions. The notion of traces provides us with what the *observable* behavior of an SIOA is. The definition below formally states what executions and traces of an SIOA are.

Definition 6.3 (Execution and traces):
An *execution fragment* α of an SIOA $\mathcal{A}$ is a non-empty (finite or infinite) sequence $s_0 l_1 s_1 l_2 \ldots$ of alternating states and actions such that

- $s_{i-1} \xrightarrow{l_i} s_i$, and
- a finite fragment α ends in a state.

α is an *execution* if $s_0 \in start(\mathcal{A})$.

Given an execution $\alpha = s_0 l_1 s_1 l_2 \ldots$ of $\mathcal{A}$, the *trace* t of α in $\mathcal{A}$, written $trace_{\mathcal{A}}(\alpha)$, is the sequence that results from removing all states from α. The *external trace* xt of α, written $xtrace_{\mathcal{A}}(\alpha)$, is the sequence that results from removing actions l_i from α such that $l_i \in int(\mathcal{A})(s_i)$ and then removing all states from the resulting sequence.

We write $s \xrightarrow{\alpha}_{\mathcal{A}} s'$ if there exists an execution fragment α of $\mathcal{A}$ starting in s and ending in s'. If this holds for $\mathcal{A}$, s' is *reachable* from s. We write α **pr** α' to denote that an execution fragment α is a *prefix* of another execution fragment α'. The same notation is used for traces. Note that the sets of traces of an SIOA are *prefix-closed*, meaning if a trace $t \cdot e$ is in the trace set for some state or action e, then t is also in the trace set.

Parallel composition. A system typically is represented by a number of SIOA, each representing an entity in the system. One way to see how they interact is by composing them together. The operation *parallel composition* provides the technical definition of how this interaction looks like.

The main idea for the parallel composition is that the composition identifies the same actions l in different SIOA and combine it as one action. Therefore, all participating SIOA perform together the transition of l.

As with I/O automata ([Lyn96, p. 207]), not all SIOA can be composed. Two SIOA that can be composed are called *compatible*. By compatible, we mean that internal actions of an SIOA $\mathcal{A}$ should not be part of the actions of another SIOA $\mathcal{A}'$. Otherwise, an internal action of $\mathcal{A}$ can force $\mathcal{A}'$ to make a transition. These SIOA

should also not produce a common output. They may, however, receive the same input mimicking situations such as multiple devices receiving the same broadcast message. On I/O automata, this check is performed only by comparing the signatures of automata, because the signature is static. For SIOA, however, this check must be performed on the states, because the signatures vary with the states. We follow the conservative approach of Attie and Lynch [AL15] which requires compatibility for all possible pairings of the states of the two SIOA being composed, instead of just checking the compatibility of the state pairs that are reachable in the execution of the composed SIOA (e.g., as in interface automata [AH01]).

Definition 6.4 (Compatible SIOA):
Let $sig = \langle in, out, int \rangle$ and $sig' = \langle in', out', int' \rangle$. sig is *compatible* with sig' iff we have:

1. $\widehat{sig} \cap int' = \emptyset$,
2. $\widehat{sig'} \cap int = \emptyset$, and
3. $out \cap out' = \emptyset$.

Let $\mathcal{A}_1, \mathcal{A}_2$ be SIOA. $\mathcal{A}_1, \mathcal{A}_2$ are *compatible* iff $\forall s_1 \in states(\mathcal{A}_1), s_2 \in states(\mathcal{A}_2)$: $sig(\mathcal{A}_1)(s_1)$ is compatible with $sig(\mathcal{A}_2)(s_2)$.

The parallel composition of two such SIOA $\mathcal{A}_1, \mathcal{A}_2$ produces an SIOA $\mathcal{A}$ where common actions of the input signature and output signature of $\mathcal{A}_1$ and $\mathcal{A}_2$ are recategorized as internal actions. The transitions of $\mathcal{A}$ are derived from the transitions of $\mathcal{A}_1$ and $\mathcal{A}_2$ such that when both are able to take a transition of the same event, then the transition is synchronized in $\mathcal{A}$. Otherwise, only one of the two SIOA makes the transition and change the state accordingly. The signature recategorization resembles the parallel composition of interface automata [AH01].

Definition 6.5 (Composition of SIOA):
Let $\mathcal{A}_1, \mathcal{A}_2$ be compatible SIOA. The *parallel composition* of these SIOA, written $\mathcal{A} = \mathcal{A}_1 \parallel \mathcal{A}_2$ is the SA consisting of the following parts:

1. A set of states $states(\mathcal{A}) = states(\mathcal{A}_1) \times states(\mathcal{A}_2)$.
2. A set of initial states $start(\mathcal{A}) = start(\mathcal{A}_1) \times start(\mathcal{A}_2)$.
3. A signature mapping $sig(\mathcal{A})$ such that for each $s = \langle s_1, s_2 \rangle \in states(\mathcal{A})$,
   (a) $in(\mathcal{A})(s) = (in(\mathcal{A}_1)(s_1) \cup in(\mathcal{A}_2)(s_2)) - out(\mathcal{A}_1)(s_1) - out(\mathcal{A}_2)(s_2)$

Definition 6.5 (Continued)

(b) $out(\mathcal{A})(s) = (out(\mathcal{A}_1)(s_1) \cup out(\mathcal{A}_2)(s_2)) - in(\mathcal{A}_1)(s_1) - in(\mathcal{A}_2)(s_2)$

(c) $int(\mathcal{A})(s) = int(\mathcal{A}_1)(s_1) \cup int(\mathcal{A}_2)(s_2) \cup ((in(\mathcal{A}_1)(s_1) \cup in(\mathcal{A}_2)(s_2)) \cap (out(\mathcal{A}_1)(s_1) \cup out(\mathcal{A}_2)(s_2)))$

4. A transition relation $steps(\mathcal{A}) \subseteq states(\mathcal{A}) \times acts(\mathcal{A}) \times states(\mathcal{A})$, such that $(\langle s_1, s_2\rangle, e, \langle s_1', s_2'\rangle) \in steps(\mathcal{A})$ if

   (a) $e \in \widehat{sig}(\mathcal{A})(\langle s_1, s_2\rangle)$

   (b) $\forall i \in \{1,2\}:$ if $e \in \widehat{sig}(\mathcal{A}_i)(s_i)$ then $s_i \xrightarrow{e}_{\mathcal{A}_i} s_i'$, otherwise $s_i' = s_i$.

It is important for the parallel composition operator to be well-defined. This means that the resulting SA is an SIOA, as shown in the following proposition.

Proposition 6.1:
Let $\mathcal{A}_1, \mathcal{A}_2$ be compatible SIOA. Then $\mathcal{A}_1 \parallel \mathcal{A}_2$ is an SIOA.

Proof:
To prove $\mathcal{A}_1 \parallel \mathcal{A}_2$ is an SIOA, we check whether the three constraints on Definition 6.2 are fulfilled. The first constraint is fulfilled by part 4(a) of Definition 6.5. The second constraint comes from the input-enabledness of $\mathcal{A}_1$ and $\mathcal{A}_2$. The third constraint is fulfilled by part 3 of Definition 6.5.

The parallel composition does not add or remove any action that is present in the signature of the SIOA operands.

Proposition 6.2:
Let $\mathcal{A}_1, \mathcal{A}_2$ be compatible SIOA. Then for each $s = \langle s_1, s_2\rangle \in states(\mathcal{A}_1 \parallel \mathcal{A}_2)$, $\widehat{sig}(\mathcal{A})(s) = \widehat{sig}(\mathcal{A}_1)(s_1) \cup \widehat{sig}(\mathcal{A}_2)(s_2)$.

Proof:
Follows from Definition 6.5.3.

Another desirable quality of a parallel composition operator is associativity, meaning that the order by which the SIOA are composed do not matter. The parallel composition operator is associative as long as we can guarantee that the SIOA share no input actions. The following proposition shows this property, assuming that the states are flattened (i.e., $\langle s_1, s_2, s_3\rangle \in states(\mathcal{A}_1 \parallel (\mathcal{A}_2 \parallel \mathcal{A}_3))$ instead of $\langle s_1, \langle s_2, s_3\rangle\rangle$).

Proposition 6.3 (Associativity of parallel composition):
Let $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ be compatible SIOA and such that for each state triple $s_1 \in states(\mathcal{A}_1)$, $s_2 \in states(\mathcal{A}_2)$, $s_3 \in states(\mathcal{A}_3)$ the pairwise input and output signatures are disjoint. Then $\mathcal{A}_1 \parallel (\mathcal{A}_2 \parallel \mathcal{A}_3) = (\mathcal{A}_1 \parallel \mathcal{A}_2) \parallel \mathcal{A}_3$.

Proof:
Let $\mathcal{A}_l = \mathcal{A}_1 \parallel (\mathcal{A}_2 \parallel \mathcal{A}_3)$ and $\mathcal{A}_r = (\mathcal{A}_1 \parallel \mathcal{A}_2) \parallel \mathcal{A}_3$. We prove that $\mathcal{A}_l = \mathcal{A}_r$ by showing that their set of states and initial states, signature mapping, and transition relations are equal. From our flattening assumption, it immediately follows that $states(\mathcal{A}_l) = states(\mathcal{A}_r)$ and $start(\mathcal{A}_l) = start(\mathcal{A}_r)$. Let $s = \langle s_1, s_2, s_3 \rangle \in states(\mathcal{A}_l)$. For simplicity, we represent the input and output signatures of the SIOA as follows:

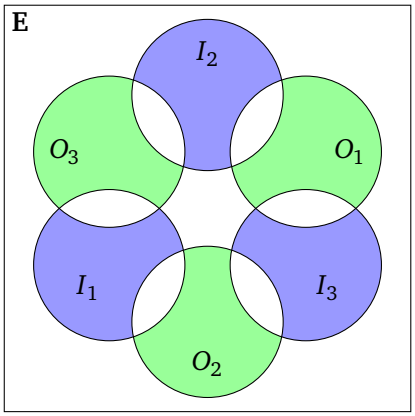
- $I_1 = in(\mathcal{A}_1)(s_1)$
- $I_2 = in(\mathcal{A}_2)(s_2)$
- $I_3 = in(\mathcal{A}_3)(s_3)$
- $O_1 = out(\mathcal{A}_1)(s_1)$
- $O_2 = out(\mathcal{A}_2)(s_2)$
- $O_3 = out(\mathcal{A}_3)(s_3)$

Following Definition 6.2 and the assumption, we gather that
- $I_1 \cap O_1 = I_2 \cap O_2 = I_3 \cap O_3 = \emptyset$,
- $I_1 \cap I_2 = I_1 \cap I_3 = I_2 \cap I_3 = \emptyset$, and
- $O_1 \cap O_2 = O_1 \cap O_3 = O_2 \cap O_3 = \emptyset$.

Then, using these assumptions, we can work out that $in(\mathcal{A}_l)(s) = in(\mathcal{A}_r)(s)$.

$$
\begin{aligned}
in(\mathcal{A}_l)(s) &= (I_1 \cup ((I_2 \cup I_3) - O_2 - O_3)) - O_1 - ((O_2 \cup O_3) - I_2 - I_3) \\
&= (I_1 \cup (I_2 - O_3) \cup (I_3 - O_2)) - O_1 - ((O_2 - I_3) \cup (O_3 - I_2)) \\
&= ((I_1 - O_1) \cup (I_2 - O_3 - O_1) \cup (I_3 - O_2 - O_1)) - ((O_2 - I_3) \cup (O_3 - I_2)) \\
&= (I_1 \cup (I_2 - O_1 - O_3) \cup (I_3 - O_1 - O_2)) - ((O_2 - I_3) \cup (O_3 - I_2)) \\
&= (I_1 \cup (I_2 - O_1 - O_3) \cup (I_3 - O_1 - O_2)) - (O_2 - I_3) - (O_3 - I_2) \\
&= (I_1 - (O_2 - I_3) - (O_3 - I_2)) \cup (I_2 - O_1 - O_3) \cup (I_3 - O_1 - O_2) \\
&= (I_1 - O_2 - O_3) \cup (I_2 - O_1 - O_3) \cup (I_3 - O_1 - O_2) \\
&= (I_1 - O_2 - O_3) \cup (I_2 - O_1 - O_3) \cup (I_3 - (O_1 - I_2) - (O_2 - I_1)) \\
&= ((I_1 - O_2 - O_3) \cup (I_2 - O_1 - O_3) \cup I_3) - (O_1 - I_2) - (O_2 - I_1) \\
&= ((I_1 - O_2) \cup (I_2 - O_1) \cup I_3) - ((O_1 - I_2) \cup (O_2 - I_1)) - O_3 \\
&= (((I_1 \cup I_2) - O_1 - O_2) \cup I_3) - ((O_1 \cup O_2) - I_1 - I_2) - O_3 \\
&= in(\mathcal{A}_r)(s)
\end{aligned}
$$

Through a similar line of reasoning we obtain $out(\mathcal{A}_l)(s) = out(\mathcal{A}_r)(s)$ and $int(\mathcal{A}_l)(s) = int(\mathcal{A}_r)(s)$. The input and output state signatures of $\mathcal{A}_l$ (and $\mathcal{A}_r$) are portrayed by the following Venn diagram as the shaded part.



Because $\widehat{sig}(\mathcal{A}_l) = \widehat{sig}(\mathcal{A}_r)$ and both $\mathcal{A}_l$ and $\mathcal{A}_r$ are SIOA, their internal signatures are the same. Hence, $sig(\mathcal{A}_l) = sig(\mathcal{A}_r)$.

Because the signatures are the same, the set of actions that are represented in the transition relation is the same on both sides. Furthermore, the synchronization part is independent on the ordering of the composition operator. Therefore, the transition relations of $\mathcal{A}_l$ and $\mathcal{A}_r$ are the same. □

The state of a composed SIOA is a Cartesian product of the states of the operands. If we consider the order of individual state elements in a composed state irrelevant (i.e., we have unordered tuples), then the parallel composition is also commutative.

Proposition 6.4 (Commutativity of parallel composition):
Let $\mathcal{A}_1, \mathcal{A}_2$ be compatible SIOA. Assuming the states of $\mathcal{A}_1 \parallel \mathcal{A}_2$ and $\mathcal{A}_2 \parallel \mathcal{A}_1$ are unordered, then $\mathcal{A}_1 \parallel \mathcal{A}_2 = \mathcal{A}_2 \parallel \mathcal{A}_1$.

Proof:
Follows immediately from the assumption and Definition 6.5.

6.2 Configuration Automata

Some actions have a side effect, namely they create new SIOA. Because SIOA represent single entities, executing these actions may give no impact on the SIOA themselves. To model this side effect, Attie and Lynch define configuration automata. A configuration automaton (CA) is a means to "keep track of the set of alive SIOA" [AL15, p. 30]. As the name suggests, CA are based on the notion of configurations: the set $\mathbb{A}$ of alive SIOA and a mapping $\mathbb{S}$ with the domain $\mathbb{A}$ such

that $\mathbb{S}(\mathcal{A})$ is the current local state of $\mathcal{A}$, for every SIOA $\mathcal{A} \in \mathbb{A}$. Based on these configurations and the underlying SIOA, we *intrinsically* derive the transitions between the configurations. What is particularly interesting with the intrinsic transitions is that we can add SIOA to the configurations. The configurations and these transitions then are weaved together to form CA. Because the creation aspect is only handled on the CA level, CA are the principle semantic objects representing the systems.

Definition 6.6 (Configuration and compatible configuration):
A *configuration* $\mathbb{C}$ is a pair $\langle \mathbb{A}, \mathbb{S} \rangle$, where

- $\mathbb{A}$ is a *finite* set of SIOA identifiers, and
- $\mathbb{S}$ maps each SIOA $\mathcal{A} \in \mathbb{A}$ to a state $s \in \mathit{states}(\mathcal{A})$.

A configuration $\langle \mathbb{A}, \mathbb{S} \rangle$ is *compatible* iff, for all $\mathcal{A}, \mathcal{A}' \in \mathbb{A}, \mathcal{A} \neq \mathcal{A}'$:

1. $\widehat{sig}(\mathcal{A})(\mathbb{S}(\mathcal{A})) \cap \mathit{int}(\mathcal{A}')(\mathbb{S}(\mathcal{A}')) = \emptyset$, and
2. $\mathit{out}(\mathcal{A})(\mathbb{S}(\mathcal{A})) \cap \mathit{out}(\mathcal{A}')(\mathbb{S}(\mathcal{A}')) = \emptyset$.

Given a configuration $\mathbb{C} = \langle \mathbb{A}, \mathbb{S} \rangle$, the functions $\mathit{auts}(\mathbb{C}) = \mathbb{A}$ and $\mathit{map}(\mathbb{C}) = \mathbb{S}$ extract the set of SIOA identifiers and the local state mapping, respectively. Furthermore, we use $(\mathcal{A}, s) \in \mathbb{C}$ as a shorthand for $\mathcal{A} \in \mathbb{A} \wedge s = \mathbb{S}(\mathcal{A})$.

A configuration is an unordered representation of pairs of an SIOA identifier and its local state. The compatibility condition reflects the compatibility conditions between SIOA applied to configurations. To establish what kind of transitions a configuration may take, its signature must be clear.

Definition 6.7 (Intrinsic signatures of a configuration):
Let $\mathbb{C} = \langle \mathbb{A}, \mathbb{S} \rangle$ be a compatible configuration. Then, the signature $\mathit{sig}(\mathbb{C}) = \langle \mathit{in}(\mathbb{C}), \mathit{out}(\mathbb{C}), \mathit{int}(\mathbb{C}) \rangle$ is the *intrinsic signature* of $\mathbb{C}$, where

- $\mathit{in}(\mathbb{C}) = (\bigcup_{\mathcal{A} \in \mathbb{A}} \mathit{in}(\mathcal{A})(\mathbb{S}(\mathcal{A}))) - \mathit{out}(\mathbb{C})$,
- $\mathit{out}(\mathbb{C}) = \bigcup_{\mathcal{A} \in \mathbb{A}} \mathit{out}(\mathcal{A})(\mathbb{S}(\mathcal{A}))$, and
- $\mathit{int}(\mathbb{C}) = \bigcup_{\mathcal{A} \in \mathbb{A}} \mathit{int}(\mathcal{A})(\mathbb{S}(\mathcal{A}))$.

The external intrinsic signature of $\mathbb{C}$ is defined as $\mathit{ext}(\mathbb{C}) = \langle \mathit{in}(\mathbb{C}), \mathit{out}(\mathbb{C}) \rangle$.

The intrinsic signature of a configuration is formed from combining the signatures of its SIOA members. While an empty configuration is a compatible configuration, it cannot make any transitions as its intrinsic signature contains no actions.

We define intrinsic transitions $\overset{l}{\Rightarrow}_{\varphi}$ that avoid transitions to incompatible configurations. An intrinsic transition takes as a parameter a set φ of SIOA identifiers that represents SIOA created by executing the transition. Attie and Lynch take the view that this parameter is given as part of the configuration automaton.

Definition 6.8 (Intrinsic transitions):
Let $\langle \mathbb{A}, \mathbb{S} \rangle$, $\langle \mathbb{A}', \mathbb{S}' \rangle$ be arbitrary compatible configurations and $\varphi \subseteq \mathit{Autids}$. Then $\langle \mathbb{A}, \mathbb{S} \rangle \overset{l}{\Rightarrow}_{\varphi} \langle \mathbb{A}', \mathbb{S}' \rangle$ iff

1. $l \in \widehat{sig}(\langle \mathbb{A}, \mathbb{S} \rangle)$,
2. $\mathbb{A}' = \mathbb{A} \cup \varphi$,
3. for all $\mathcal{A} \in \mathbb{A}' - \mathbb{A} : \mathbb{S}(\mathcal{A}) \in \mathit{start}(\mathcal{A})$,
4. for all $\mathcal{A} \in \mathbb{A}$: if $l \in \widehat{sig}(\mathcal{A})(\mathbb{S}(\mathcal{A})) \wedge \mathbb{S}(\mathcal{A}) \overset{l}{\rightarrow}_{\mathcal{A}} s$, then $\mathbb{S}'(\mathcal{A}) = s$, otherwise $\mathbb{S}'(\mathcal{A}) = \mathbb{S}(\mathcal{A})$.

An intrinsic transition is defined such that the executed action is part of the intrinsic signature (Constraint 1) and is an actual transition of some SIOA in the configuration (Constraint 4). The second constraint states that the transition may result in a configuration with additional SIOA as given in the parameter. These newly created SIOA are randomly assigned initial states (Constraint 3).

Note that the definitions of intrinsic signatures and transitions are only given for compatible configurations. Executing an action on a compatible configuration may lead incompatible configurations. There are two possibilities how this may happen:

- An action which involves two or more SIOA in the configuration may cause these SIOA to move to states with some common output actions.
- A new SIOA is created with an initial state whose signature is incompatible with the signature of the existing SIOA.

In Chapter 7, we show that in the actor setting, these possibilities cannot occur.

The intrinsic signatures and transitions are two main ingredients to define CA. What we need to add now are the SIOA themselves. The following definition shows how they are combined together as CA.

Definition 6.9 (Configuration automata):
A *configuration automaton* $\mathcal{C}$ is a triple $\langle sioa(\mathcal{C}), config(\mathcal{C}), created(\mathcal{C})\rangle$ where

- $sioa(\mathcal{C})$ is an SIOA;
  (The parts of this SIOA are abbreviated to $states(\mathcal{C}) = states(sioa(\mathcal{C}))$, $start(\mathcal{C}) = start(sioa(\mathcal{C}))$, etc., for brevity.)
- a configuration mapping $config(\mathcal{C})$ with domain $states(\mathcal{C})$ such that for all $x \in states(\mathcal{C})$, $config(\mathcal{C})(x)$ is a compatible configuration; and
- for each $x \in states(\mathcal{C})$, a mapping $created(\mathcal{C})(x)$ with domain $\widehat{sig}(\mathcal{C})(x)$ such that for all actions $l \in \widehat{sig}(\mathcal{C})(x)$, $created(\mathcal{C})(x)(l) \subseteq Autids$,

such that the following constraints are satisfied:

1. If $x \in start(\mathcal{C})$ and $(\mathcal{A}, s) \in config(\mathcal{C})(x)$, then $s \in start(\mathcal{A})$.
2. If $(x, l, x') \in steps(\mathcal{C})$ then $config(\mathcal{C})(x) \stackrel{l}{\Rightarrow}_{\varphi} config(\mathcal{C})(x')$, where $\varphi = created(\mathcal{C})(x)(l)$.
3. If $x \in states(\mathcal{C})$ and $config(\mathcal{C})(x) \stackrel{l}{\Rightarrow}_{\varphi} \mathbb{C}$ for some action l, $\varphi = created(\mathcal{C})(x)(l)$, and a compatible configuration $\mathbb{C}$, then $\exists x' \in states(\mathcal{C})$ such that $config(\mathcal{C})(x') = \mathbb{C}$ and $(x, l, x') \in steps(\mathcal{C})$.
4. For all $x \in states(\mathcal{C})$
   (a) $in(\mathcal{C})(x) = in(config(\mathcal{C})(x))$
   (b) $out(\mathcal{C})(x) = out(config(\mathcal{C})(x))$
   (c) $int(\mathcal{C})(x) = int(config(\mathcal{C})(x))$

A configuration automaton consists of an SIOA that acts as a configuration controller, a mapping from the state of the control SIOA to compatible configurations, and a mapping that states which SIOA is created after executing a transition. We do not specify this control SIOA, instead it is derived from the SIOA present in the configurations, provided the constraints are fulfilled.

The constraints placed on a CA guarantee the following. In an initial configuration, each SIOA in the configuration must be mapped to an initial state of that SIOA (Constraint 1). Constraint 2 uses the intrinsic transitions to describe the transition of the configuration automaton. This is strengthened by the third constraint that the transition relation of CA must include all intrinsic transitions the CA can do. Unlike the basic I/O automaton model, this constraint means that the

states themselves contain information about their successor states. Constraint 4 states that the signature of a state x of the CA must be equal to the signature of its corresponding configuration. This constraint may be adjusted similar to the original definition [AL15] when we define some operators on the CA.

The notions of executions and traces for CA remain the same as those for SIOA, because the main representative of a CA is an SIOA. Consequently, the trace sets are also prefix-closed.

Definition 6.10 (Executions and traces of configuration automata):
α is an execution fragment (execution) of a configuration automaton $\mathcal{C}$ iff it is an execution fragment (execution) of $sioa(\mathcal{C})$. t is a trace of $\mathcal{C}$ if it is a trace of $sioa(\mathcal{C})$. The sets of executions, traces and external traces of $\mathcal{C}$ are denoted by $execs(\mathcal{C})$, $traces(\mathcal{C})$ and $xtraces(\mathcal{C})$, respectively.

6.3 Discussion

The DIOA model as proposed by Attie and Lynch is designed to represent mobile agents. As a result, the DIOA model is very general, where the signatures may dynamically change. For example, an input action on one state may become an output action on another state. Message broadcasting to multiple entities through a single output action is also supported, as implicitly rendered possible by the intrinsic transitions. The intrinsic transition definition in [AL15] is actually even more general than Definition 6.8 because it accommodates the *removal* of no longer active SIOA. A mobile phone, for example, that leaves a mobile network can be captured by this more general definition. Because we do not concern ourselves with the explicit removal of actors, we opt for a simpler definition.

Similarly, Attie and Lynch need their notion of traces to only deal with external actions. Furthermore, the traces also include the external signatures of the SIOA. These two elements are sufficient to allow traces to become the externally visible behavior of an SIOA. Because we are using the SIOA to model the behavior of actors, these complications are not necessary. In fact, it is necessary to have (some of) the internal actions in the traces, as they enable the verification of the implementation based on the denotational semantics.

The definition for parallel composition is slightly different from Attie and Lynch's on the output signature aspect. In Definition 6.5, output actions that are input actions of the input state signatures are removed from the output state signatures on the composed SIOA. We consider such actions as being recategorized as internal actions. Attie and Lynch refrain from recategorizing output actions, keeping

an action that is part of the output state signature of one of the SIOA operands as an output action in the composed SIOA. This difference is deliberately put in there because the actor model has no direct support for message broadcasting. Furthermore, the definition for parallel composition originally accepts a variable number of SIOA as the operands. The motivation for such a general version is to enable the composition of "a finite number of large systems" (such as the cell towers for transmitting mobile signals). This intention is not present here, so we opt for the binary version.

Similar to SIOA, we can also define a parallel composition operator on CA. Since this operator is not needed in the adaptation for the actor model, interested readers are referred to Attie and Lynch's report [AL15, Section 5.1].

In the following chapters, we see how this DIOA model is adapted to suit the actor model and the component notion we have described in Chapter 5. The adaptation of SIOA to single actors is defined in Chapter 7, while the adaptation of CA to actor systems is defined in Chapter 8. Together, they constitute an interpretation of the actor model in a dynamic automaton model. The inclusion of the component notion is described in Chapter 9.

CHAPTER 7

Class Behavior Representation

In αABS the behavior of an actor is described by a class. To model this behavior, we introduce an adaptation of SIOA called *actor automata* (AA) that fits well with αABS. AA are designed to capture the characteristics of an actor. We attain this goal by ensuring that the traces of AA are well-formed.

The main idea of the adaptation is to embed the desired characteristics as deep as possible into AA. For example, the AA model utilizes the possibility to change the state signatures of SIOA by removing a set of input events from a state signature, once an input event using the same future is executed. To be able to manipulate the signatures, information about received futures, sent method calls and other known actors, among others, is encoded in the states. Through such embedding, we ensure that the AA model produces only well-formed traces. Furthermore, an AA can be specified concentrating on the specific behavior represented by a particular class.

Chapter outline. This chapter begins with a formal representation of AA in Section 7.1. Section 7.2 describes the properties of AA. Finally, Section 7.3 provides a short discussion on our encoding of the general characteristics of an actor. The complete adaptation of DIOA for the actor model is provided in the next chapter.

7.1 Model

An AA models the behavior of an actor $a \in \mathbf{A}$ by refining the state space of a signature automaton to deal with the generation of unique futures, the generation of new actor names, the blocking to wait certain future resolutions and exposure of other actors to a. Because all AA representing actors of the same *class* have the same set of states and transitions save for the name of the actors, it is only natural to supply the actor name as a parameter to the AA. This name acts as a convenient reminder of which actor the automaton is representing as well as a means to classify the state signatures. This parameterization can be removed by enriching the state with a self reference that stays constant in each transition. To refer to specific parts of a state of an AA we use the universe of variables $\mathbf{V}$.

Transitions of AA are labeled by parameterized events, where the parameter is the actor name.

Definition 7.1 (Parameterized events):
A parameterized event *ev* is an event *e* where *caller*(*e*), *target*(*e*) and some values of *param*(*e*) can be the *this* variable.

The instantiation of *ev* is represented by *ev*(*a*), where *a* is an actor name of the specified class. As we use the events only in an instantiated context, we abuse the notation in the following definitions where for each event $e \in \mathbf{E}$ as defined in Definition 4.1 appearing as a transition label is actually *ev*(*this*).

The complete definition of AA is given by Definition 7.2. For convenience, we let **U** contain a special member $\perp$ that cannot be generated by any actor.

AA are instances of SA with the states now containing information specific to the need to model class-based actors with method returns. The automaton is parameterized with the actor name, allowing a single representation of actors of the same class. For simplicity, we allow the name parameter of an actor automaton to be dropped when it is clear from the context (i.e., $\mathcal{A}$ is used instead of $\mathcal{A}(a)$ when it is clear). The actor name can be retrieved by using the function *names* that maps an AA (identifier) to an actor ($names(\mathcal{A}(a)) = \{a\}$[1]).

State variables. The variable *cons* indicates that the class constructor has been executed if it is set to true. The variables buf_c and buf_r act as the buffer of an actor that store input events. The variable buf_c stores the incoming calls, while buf_r stores the method returns. The buffer is divided into two to enable multiple fetching of resolved future values. When an actor reacts to an incoming method call, a task is created. As αABS allows cooperative multitasking, the model needs to store the *tasks* an actor is currently working at. Only one task may be actively processed by the actor and this requirement is represented in the automata by the variable *release*, which is assigned to true if the actor has no active task (i.e., currently at the release point). Any transition can introduce a release point, when it represents receiving an input. This flexibility is necessary to model cooperative multitasking. To synchronize with other actors when processing a task, an actor can block to wait for the result of a method call. This blocking is represented in the model by storing the *blocked future* in *blkFut* that needs to be resolved. We use the symbol $\perp$ to represent the state where no blocking is being placed.

Identifying events an actor can receive and generate requires the information on the actors it knows (*known*), the pairs of future and target actor of the call events

[1] A set notation is used to provide better compatibility with the configuration automata.

it has received (*rcvFutTgt*), the futures it has generated (*futGen*) and the actors it has created (*nameGen*). Both the future and actor name generators ensure only fresh identifiers are produced. The events the actor has generated so far are stored in t_{gen}. For AA, it is actually not necessary to store the target actor of incoming call events, however this allows compatibility on the component level. In addition, the state stores the calls the actor made to other actors that have not been replied yet (*outCalls*). Internal variables used by the actor are categorically represented by the variable *ints*. The parameters passed on when creating the actor are categorically represented by the variable *params*. The precise description of the internal variables and the parameters is presented in the specification of AA, where its usage to control the actions of the actors is described.

Transition constraints. The constraints show the dynamics of each of these state elements with respect to the signatures and the transition relation. Constraint A1 states that every transition happens by executing an event of the state signature. This constraint is a refined version of the original first constraint of SIOA, which is required to show that an AA is indeed an SIOA.

Initial state constraint. Constraint A2 states the generic property of initial states. In an initial state, all sets are initialized to empty set, except for the known actor set, because it may contain actor names that are part of the class parameters, and no blocking is being placed. The initial value of the variable *cons* is left open to the specification of the AA.

State signature constraints. The state signatures of non-initial states are characterized by Constraints A3 to A5. These constraints can be summarized as follows. Reaction events are internal events, as they are only observable by the actors that produce them. An emittance event generated by the actor is an output event, except for a self call, which is classified as an internal event. An emittance event generated by other actors and targeted to the actor is an input event. The message of every emittance event must be in the set of allowed messages governed by the class of the represented actor. The following explanation gives more details on how the signatures evolve.

Constraint A3 ensures that events in the input state signatures have characteristics of input events: emittance events generated by other actors. If the input event is a method call, the future of that input event has never been received by the actor. If it is a return, its event core corresponds to the event core of an outgoing call that has not been returned yet.

Definition 7.2 (Actor automata):
A parameterized SA $\mathcal{A}(this) = \langle states(\mathcal{A}), start(\mathcal{A}), sig(\mathcal{A}), steps(\mathcal{A})\rangle$ with the following description:

- $states(\mathcal{A})$ is a map with a fixed domain $V \subseteq \mathbf{V}$ denoting the variables stored by the actor. V includes the following fixed variables: *cons*, buf_c, buf_r, *tasks*, *blkFut*, *release*, *known*, *rcvFutTgt*, *outCalls*, *futGen*, *nameGen* and t_{gen}, representing whether the class constructor has been executed ($\mathbf{B}$), a call event buffer (of type $2^{\mathbf{E}}$), a return event buffer ($2^{\mathbf{E}}$), the set of tasks ($eCore(2^{\mathbf{E}})$), whether the actor is at a release point ($\mathbf{B}$), the blocking future ($\mathbf{U}$), the set of known actors ($2^{\mathbf{A}}$), the set of pairs of received future and target ($2^{\mathbf{U}\times\mathbf{A}}$), the set of outgoing calls ($eCore(2^{\mathbf{E}})$), the future generator ($2^{\mathbf{U}}$), the actor name generator ($2^{\mathbf{A}}$) and the generated trace ($Seq\langle\mathbf{E}\rangle$), respectively. The read-only class parameters are stored under the variable *params*. Other variables are deemed internal and grouped together under *ints*;
- a non-empty set of initial states $start(\mathcal{A}) \subseteq states(\mathcal{A})$;
- a signature mapping $sig(\mathcal{A})$ where for each state $s \in states(\mathcal{A})$, $sig(\mathcal{A})(s) = \langle in(\mathcal{A})(s), out(\mathcal{A})(s), int(\mathcal{A})(s)\rangle$ and $\widehat{sig}(\mathcal{A})(s) \subseteq \mathbf{E}$;
- a transition relation $steps(\mathcal{A}) \subseteq states(\mathcal{A}) \times acts(\mathcal{A}) \times states(\mathcal{A})$;

is an *actor automaton* representing an actor of name *this* when it satisfies the following constraints where $C_{this} = class(this)$:

A1. $\forall (s, e, s') \in steps(\mathcal{A}) : e \in \widehat{sig}(\mathcal{A})(s)$

A2. $\forall s \in start(\mathcal{A}) : s(buf_c) = s(buf_r) = s(tasks) = s(futGen) = \emptyset \wedge$
$s(rcvFutTgt) = s(nameGen) = s(outCalls) = \emptyset \wedge this \in s(known) \wedge$
$s(t_{gen}) = [\,] \wedge s(blkFut) = \bot \wedge s(release)$

A3. $\forall s \in states(\mathcal{A}) : in(\mathcal{A})(s) =$
$$\left\{ e \;\middle|\; \begin{array}{l} isEmit(e) \wedge msg(e) \in aMsg(C_{this}, in) \wedge \\ (isCall(e) \implies target(e) = this \wedge caller(e) \neq this \;\wedge \\ \qquad\qquad \langle fut(e), this\rangle \notin s(rcvFutTgt)) \wedge \\ (isRet(e) \implies eCore(e) \in s(outCalls)) \end{array} \right\}$$

Definition 7.2 (Continued)

A4. $\forall s \in states(\mathcal{A}) : out(\mathcal{A})(s) =$

$$\left\{ e \;\middle|\; \begin{array}{l} isEmit(e) \wedge acq(e) \subseteq s(known) \wedge msg(e) \in aMsg(C_{this}, out) \wedge \\ (isRet(e) \implies eCore(e) \in s(tasks) \wedge caller(e) \neq this) \wedge \\ (isCall(e) \implies fut(e) \notin s(futGen) \wedge caller(e) = this \wedge \\ \qquad target(e) \neq this) \wedge \\ (isCreate(e) \implies target(e) \notin s(nameGen) \wedge caller(e) = this) \end{array} \right\}$$

A5. $\forall s \in states(\mathcal{A}) : int(\mathcal{A})(s) =$

$$\left\{ e \;\middle|\; \begin{array}{l} (isReact(e) \implies emitOf(e) \in s(buf_c) \cup s(buf_r)) \wedge \\ (isEmit(e) \implies isMethod(e) \wedge msg(e) \in aMsg(class(this), int) \wedge \\ \quad caller(e) = target(e) = this \wedge acq(e) \subseteq s(known) \wedge \\ \quad (isCall(e) \implies fut(e) \notin s(futGen)) \wedge \\ \quad (isRet(e) \implies eCore(e) \in s(tasks))) \end{array} \right\}$$

A6. $\forall s \in states(\mathcal{A}) : \forall e \in in(\mathcal{A})(s) : \exists s' \in states(\mathcal{A}) : (s, e, s') \in steps(\mathcal{A}) \wedge$

$$s' = s \left[\begin{array}{l} buf_c \mapsto s(buf_c) \cup \{e \mid isCall(e)\}, buf_r \mapsto s(buf_r) \cup \{e \mid isRet(e)\}, \\ rcvFutTgt \mapsto s(rcvFutTgt) \cup \{\langle fut(e), this\rangle \mid isCall(e)\}, \\ outCalls \mapsto s(outCalls) - \{eCore(e) \mid isRet(e)\} \end{array} \right]$$

A7. $\forall (s, e, s') \in steps(\mathcal{A}) : \exists s'' : isReact(e) \implies s(release) \wedge$
$diffOn(s, s'', \{buf_c, buf_r, known, release, ints, t_{gen}\}) \wedge s''(t_{gen}) = s(t_{gen}) \cdot e \wedge$
$s''(buf_c) = s(buf_c) - \{emitOf(e)\} \wedge s''(known) = s(known) \cup acq(e) \wedge$
$(s(blkFut) \neq \bot \implies isRet(e) \wedge fut(e) = s(blkFut) \wedge$
$\qquad s' = s''[blkFut \mapsto \bot]) \wedge$
$(s(blkFut) = \bot \implies s' = s''[tasks \mapsto s''(tasks) \cup \{eCore(e) \mid isCall(e)\}])$

A8. $\forall (s, e, s') \in steps(\mathcal{A}) : \exists s'', s''' : isEmit(e) \wedge e \in out(\mathcal{A})(s) \cup int(\mathcal{A})(s) \implies$
$s(blkFut) = \bot \wedge diffOn(s, s''', \{ints, blkFut, release, t_{gen}\}) \wedge$
$s'''(t_{gen}) = s(t_{gen}) \cdot e \wedge$
$(isRet(e) \implies eCore(e) \in s'''(tasks) \wedge$
$\qquad s''(tasks) = s'''(tasks) - \{eCore(e)\}) \wedge$
$(isCall(e) \implies s''(futGen) = s'''(futGen) \cup \{fut(e)\}) \wedge$
$(isCreate(e) \implies s''(known) = s'''(known) \cup \{target(e)\} \wedge$
$\qquad s''(nameGen) = s'''(nameGen) \cup \{target(e)\}) \wedge$
$(e \in int(\mathcal{A})(s) \implies (isCall(e) \wedge s' = s''[buf_c \mapsto s''(buf_c) \cup \{e\}]) \vee$
$\qquad (isRet(e) \wedge s' = s''[buf_r \mapsto s''(buf_r) \cup \{e\}])) \wedge$
$(e \in out(\mathcal{A})(s) \implies$
$\qquad s' = s''[outCalls \mapsto s''(outCalls) \cup \{eCore(e) \mid isCall(e)\}])$

The output state signatures are described in Constraint A4 by looking at the types of the events. Only emittance events are contained in the output state signature and for each emittance event, its acquaintance must be known in that state. A return event must be the result of finishing one of the tasks originating from other actors. If it is a method call event, then the emittance event must have a future that is constructed from the actor name and the future generator. The future generator behaves non-deterministically, but since it stores the generated future identifiers, no duplicate will be generated. The event must also be targeted to a different actor from *this* and the method call must be part of the interface of that other actor. If it is a creation event, the actor decides non-deterministically, as with the future generator, the name of the new actor. Because an actor name is hierarchically structured, the created actor cannot have the same name as *this*.

The internal state signatures contain all reaction events, self calls and their returns as described by Constraint A5. The additional condition on method return events reflects on the existence of corresponding internal tasks the actor is handling.

Transition relation constraints. Constraints A6 to A8 deal with the restriction on the transition relation. Constraint A6 ensures that the input-enabled property of SIOA (Constraint 2 of Definition 6.2) is preserved. The effect of executing an input event means storing it in the actor's buffer, mimicking the buffering of incoming messages. If the input event is a method call, then its future and target actor (which in this case is always the represented actor *this*) are recorded as received. Otherwise, the event core of the outgoing call matching the event core of method return input event is removed from the state. This state modification ensures that an actor cannot respond to multiple events with the same future. Normally this property is ensured on the global level by having each actor generating unique futures. We localize this property to allow well-formedness criteria on the resulting traces of single actors. No other state information is changed.

The last two constraints deal with non-input events, i.e., the events generated by the actor. Constraint A7 states that executing a method call reaction event means removing the corresponding emittance event from the call buffer. Executing a method return reaction event does not remove the corresponding emittance event from the return buffer, enabling multiple fetching of the resolved value. As this reaction event is generated by the actor, it is appended to t_{gen}. A reaction event can happen only when an actor is at a release point. Because of the constraints on the state signatures, it is guaranteed that the corresponding emittance event exists in the buffer. This emittance event is then removed from the buffer. Because this transition happens when the actor actually obtains the content of the event

parameters, the known actor set is enlarged by the acquaintance in the event. When the actor is in blocking mode waiting for a return event, the corresponding reaction event is prioritized. No other event may be processed in the mean time, except for the input events (due to the input-enabledness requirement). The blocking is then resolved by executing the transition. Otherwise, a method call reaction event is transformed into a task the actor has to do. Changes pertaining the release point and the internal variables are allowed, but their specifications are left open.

Constraint A8 deals with a transition of an output or internal emittance event. An emittance may occur only when the actor is not in blocking mode. When it occurs, the event is appended to t_{gen}. A return event indicates that a task is finished. A method call event requires a new future to be generated, while a creation event requires a new actor name to be generated. The generated future or name is guaranteed to be locally unique due to the restriction placed by Constraints A4 and A5 on the output and internal signatures. As the creator, the actor then knows the created actor. If the event is an internal event, it goes directly to the call or return buffer. Otherwise, the outgoing calls are updated accordingly. Left open to the specification are the changes on the internal variables, whether the actor blocks and whether the current task is now suspended or completed, leading to a release point.

The constraints on the state signatures mark a difference between the class-based approach and the update-based approach. An update-based actor may radically change its input interface according to its state. Thus, the actor must be able to accept any input message although it may not react to them at all.

This definition can be extended to accommodate a creation reaction event (that is, fully adopting the event types of Din et al. [DDJO12]), by distinguishing between initial and non-initial states. The creation reaction events allow a direct support for class constructors. The creation emittance event is stored in the buffer of the initial state and allows only its corresponding reaction event to be present in the signature of that initial state. The constraints on the input, output and internal state signatures as described above are then applied only to non-initial states. Should a specific scheduling strategy for retrieving input events from the buffer be desired, the type of the variable *release* and related constraints can be refined accordingly.

Example 7.1.1:
The AA of the `Server` and `Worker` classes described in Chapter 2 are specified in Figures 7.1 and 7.2, respectively. The specification has a similar format to the I/O automata "precondition effect" pseudocode [Lyn96, Chapter 8], tailored for our

Allowed messages

`Server`():
provided: `Value serve(Query` q`)`
required: `new Worker()`
`Value` w : `do(Query` q`)` $class(w) = \texttt{Worker}$
internal: none

State

$\underline{\text{futPair}} \subseteq \mathbf{U} \times \mathbf{U}$, **initially** $\emptyset$

Actions

$u \twoheadrightarrow this$: `serve(`q`)` $\cdot\ this \rightarrow w$: `new Worker()` $\cdot\ u' \rightarrow w$: `do(`q`)`
pre: true
state: $\underline{\text{futPair}} := \underline{\text{futPair}} \cup \{\langle u, u' \rangle\}$

$u' \twoheadleftarrow w$: `do` $\triangleleft\ v \cdot u \leftarrow this$: `serve` $\triangleleft\ v$
pre: $\langle u, u' \rangle \in \underline{\text{futPair}}$
state: $\underline{\text{futPair}} := \underline{\text{futPair}} - \{\langle u, u' \rangle\}$

Figure 7.1.: A specification of actor automaton `Server`(*this*) for `Server` class

actor setting. The specification has three parts defining the allowed messages of the actor, its internal variables and its actions.

- The **Allowed messages** section specifies the class parameters (if any), the incoming calls that the actor can handle (the **provided** interface), the output messages it may produce in order to respond to incoming calls (the **required** interface), and the internal messages that may occur. This classification of allowed messages is transformed into the signature of the AA semantics of the specification.
- The **State** section specifies the internal variables, their types and their initial value using the **initially** keyword. The domain of the data type `D` is assumed by **D**(`D`).
- The **Actions** section specifies the sequence of events an actor takes until it reaches a release point or it actively blocks to retrieve the resolved value of a future. In other words, the specification covers the produced internal and output events from one release or blocking point to another release or blocking point. Consequently, there can be at most one reaction event in the sequence.

Allowed messages

`Worker`():
provided: `Value do(Query q)`
required: `new Worker()`
`Value w : do(Query q)` $class(w) = \texttt{Worker}$
internal: none

State

$\underline{\text{futTriple}} \subseteq \mathbf{U} \times \mathbf{U} \times \mathbf{D}(\texttt{Query})$, **initially** $\emptyset$

Actions

$u \twoheadrightarrow this : \texttt{do}(q) \cdot this \rightarrow w : \texttt{new Worker}() \cdot u' \rightarrow w : \texttt{do}(\texttt{restQuery}(q))$
pre: $\texttt{querySize}(q) > 1$
state: $\underline{\text{futTriple}} := \underline{\text{futTriple}} \cup \{\langle u, u', q\rangle\}$

$u' \twoheadleftarrow w : \texttt{do} \triangleleft v \cdot u \leftarrow this : \texttt{do} \triangleleft \texttt{merge}(\texttt{compute}(\texttt{firstQuery}(q)), v)$
pre: $\langle u, u'\rangle \in \underline{\text{futTriple}}$
state: $\underline{\text{futTriple}} := \underline{\text{futTriple}} - \{\langle u, u', q\rangle\}$

$u \twoheadrightarrow this : \texttt{do}(q) \cdot u \leftarrow this : \texttt{do} \triangleleft \texttt{compute}(q)$
pre: $\texttt{querySize}(q) = 1$
state: no change

Figure 7.2.: A specification of actor automaton `Worker`(*this*) for `Worker` class

If the sequence of events contains a reaction event, this event must appear as the first event of the sequence. The specification also defines the change on the values of the internal variables as marked by the **state** keyword. After the sequence of events is generated, the state of the internal variables has to fulfill the given specification. We can also specify when a blocking is desired by stating $blkFut = u$ where u is the future that the actor should wait to be resolved. As the specification technique name suggests, it is possible to control which sequence an actor can perform next. This *precondition*, marked by the **pre** keyword and defined over the internal states, specifies the conditions under which the sequence of events is permitted to occur.

Each of the `Server` and `Worker` class specifications has these three parts. Comparing the class implementation with the specification provides some hints how an AA represents an actor. The **Allowed messages** section of the class specifications

represents the possible method calls and actor creation an instance of a class can handle or generate. The **provided** part of the class specifications resembles the methods of the interface the classes implement. For example, the `serve` method in the `Server` class specification is part of the `IServer` interface. The **required** part of the class specifications resembles the method call and create statements in the implementation. A `Worker` actor can create a new worker and call this new worker with the `do` method. As there is no need to produce some internal behavior, the **internal** part of both specifications is left empty.

The internal state of an instance of a `Server` class is defined as a set of pairs of futures. A server actor needs to associate the future of a request with the future it generates when calling the worker to compute the request query. In the implementation, this control task is done implicitly by the underlying operational semantics. Because AA do not provide such support, this control must be represented explicitly in the specification. Similarly, the `Worker` class specification uses a set of triples $\langle u, u', q \rangle$. The added query q is stored so that a worker actor can complete the computation when it returns the result.

The **Action** section of the class specifications follows the description of our server requirement (Section 2.1). It also illustrates how the internal state is being used to control the connection between the incoming calls and the calls generated by the actors.

In Section 10.1, we provide the semantics of the class specifications in terms of AA. Roughly, the semantics translates a class specification into an AA by

- populating the signatures based on the **Allowed messages** section,
- populating *ints* by the variables defined in the **State** section,
- transferring the sequence transition described in the **Action** section into single transitions of the AA, and
- completing the state space, the state signatures and the transition relation of the AA as required by the constraints of Definition 7.2.

Thus, from the `Server` specification, the corresponding AA $\mathcal{A}(s)$ can produce the following trace of a server actor s.

$$u_1 \rightarrow s : \texttt{serve}(q_1) \cdot u_1 \twoheadrightarrow s : \texttt{serve}(q_1) \cdot s \rightarrow w_1 : \texttt{new Worker}() \cdot u_1' \rightarrow w_1 : \texttt{do}(q_1) \cdot$$
$$u_1' \leftarrow w_1 : \texttt{do} \triangleleft v_1 \cdot u_1' \twoheadleftarrow w_1 : \texttt{do} \triangleleft v_1 \cdot u_1 \leftarrow s : \texttt{serve} \triangleleft v_1$$

The `serve` call and `do` return emittance events are generated by the AA as required by Constraint A6. △

7.2 Properties

In this section we look into several properties of AA. A fundamental property is that AA are SIOA as stated by the following proposition. This has two implications. First, it allows us to talk about SIOA that represent the behavior of single actors, while reusing the definitions of executions of traces that come with SIOA. Second, it enables us to use operations on SIOA such as the parallel composition operator.

Proposition 7.1:
An actor automaton $\mathcal{A}$ is an SIOA.

Proof:
$\mathcal{A}$ is a signature automaton. Thus, only the constraints of Definition 6.2 need to be checked. The first constraint follows directly from Constraint A1, whereas the second constraint on input-enabledness follows from constraint A6. The disjointness of the state signatures as expressed by the third constraint comes from Constraints A3 to A5, where each event is classified as input, output or internal based on its caller, target, future and type. □

We expect that the traces of an actor automaton to be well-formed (Definition 4.3). The following lemma states that generated traces of an actor automaton $\mathcal{A}(a)$ are well-formed. The idea behind the proof of this lemma is that the constraints of AA are designed to capture the well-formedness properties.

Lemma 7.1 (Well-formedness of generated traces of actor automaton):
Let $\mathcal{A}(a)$ be an actor automaton of actor a. Let $\alpha \in \mathit{execs}(\mathcal{A}(a))$ be an execution of $\mathcal{A}(a)$ such that t is the derived trace of α. Then, t is well-formed with respect to $A = \{a\}$.

Proof:
The proof follows from the constraints on actor automata.

Property 1: If there is no event in α with the future u, the property holds. Assume the projection of the trace to the future u results in events $t' = e_1 e_2 e_3 \ldots$. The first event of this projected trace t' is a method call emittance event. By Constraint A3, a method return event is present in the input signature, but only if the corresponding event core is in the set of tasks. Similar argument is fulfilled for the output and internal signatures by means of Constraints A4 and A5. However, initially the set of tasks is empty. For an event core to be in the set of tasks, the corresponding input event has to be in the buffer (Constraints A7 and A8). Thus, the premise holds. We proceed by proving what happens after the method call emittance event is received.

Case *internal call*: Let e_1 be an internal method call event, that is, $gen(fut(e_1)) = a \wedge target(e_1) = a$. By Constraint A8, e_1 is placed into the buffer. Because the same future as the one used for e_1 cannot be used in other method call events produced by the actor (Constraint A4, A5 and A8), we exclude the possibility of other method call events with the same future appearing in the projection of the trace. Consequently, when $|t'| > 1$, e_2 must be a reaction event of e_1 (Constraint A7). The corresponding event core of e_2 is inserted into the actor task set. When $|t'| > 2$, the only option is to finish processing the task and return the result of executing the method (Constraint A8). Following Constraint A8, this return event e_3 is stored in the buffer. The only extension left for the projected trace is (a sequence of) the reaction event of e_3. Thus, the property holds.
Case *incoming method call*: Let e_1 be a method call emittance input event, that is, $caller(e_1) \neq a$. By Constraint A6, e_1 is placed into the buffer and the future of e_1 (and a) is stored in the set of received futures, effectively removing all events with the same future from the input signature (Constraint A3). When $|t'| > 1$, e_2 must be a reaction event of e_1, because no further input with the same future may appear and no task with the same future is present in the set of tasks of the actor (Constraint A7). The corresponding event core of e_2 is inserted into the actor task set. As the input signature excludes method call events whose futures have been received previously, the automaton never returns back to the state with an input signature that contains e_1. When $|t'| > 2$, the only option is to finish processing the task and return the result of executing the method (Constraint A8). Then it follows that the projected trace may only contain a method call event possibly followed by the corresponding method return event.
Case *outgoing method call*: Let e_1 be a method call emittance output event, that is $caller(e_1) = a \wedge target(e_1) \neq a$. Following Constraint A8, such an event only appears in the output signature. Because the generated future is stored, the same future will never be generated again (Constraints A4 and A5). The input signature is updated accordingly to allow for the corresponding method return events. For the projected trace to proceed, only a corresponding method return event may appear as an input. That is, e_2 is of the form $u \leftarrow b.mtd \triangleleft v$ for some result v. Following Constraint A6, this input event is stored in the buffer. The only extension left for the projected trace is (a sequence of) the reaction event of e_2. Thus, the property holds.

Property 2: Let α be such that $trace(\alpha) = t \cdot e$ where e is an emittance event and generated by a. For any transition (s, e', s') such that $s\ e'\ s'$ is a part of α, the set of known actors grows monotonically ($known(s) \subseteq known(s')$) following Constraints A6 to A8. If e' is a non-reaction internal event (a.k.a., self calls and their returns), the set of known actors does not change ($known(s) = known(s')$)

because the acquaintance of e' must already be part of the set of known actors (Constraint A8). As e is appears in $trace(\alpha)$ and generated by a, there must be a transition $(s_e, e, s'_e) \in steps(\mathcal{A})$ taken to make that event appear in e. As Property 1 holds, e can only appear once in $trace(\alpha)$, meaning it cannot appear in t. For transition (s_e, e, s'_e) to appear, Constraint A8 must be fulfilled, particularly $acq(e) \subseteq known(s_e)$. Similarly, every consecutive triple (s, e', s') in α must obey the Constraints A4 and A5, maintaining the monotonicity property. As t contains input events that may not be reacted upon, $known(s_e) \subseteq acq(t) \cup \{a\}$. Thus, $acq(e) \subseteq acq(t) \cup \{a\}$.

Property 3: Follows directly from Constraints A8. □

The well-formedness property does not filter out degenerate cases such as the environment may send the actor a a reference to an actor that is not yet created (or even an actor that is supposed to be created by a!). While it is possible to prune down more degenerate cases by refining the AA definition, the removal of such cases is better done on the configuration level, where complete information of the system is available.

The parallel composition operator is crucial for our verification effort, because it enables us to relate the composed SIOA with some more general SIOA representing the components. Through the following lemmas, we show that different AA are indeed *compatible*, which fulfills the condition for the parallel composition operator. The main reason why they are compatible is because the actions used in the AA (i.e., the events) involve at most two actors.

Lemma 7.2 (Disjoint actor automaton signature):
Let $\mathcal{A}$ be an actor automaton. We define $in(\mathcal{A})$ to be $\bigcup_{s \in states(\mathcal{A})} in(\mathcal{A})(s)$, and similarly for $out(\mathcal{A})$ and $int(\mathcal{A})$. Then,

$$in(\mathcal{A}) \cap out(\mathcal{A}) = in(\mathcal{A}) \cap int(\mathcal{A}) = out(\mathcal{A}) \cap int(\mathcal{A}) = \emptyset \ .$$

Proof:
First we show that any event in the input signature never belongs to the other two signatures. Assume $e \in in(\mathcal{A})$. By Constraints A3, e must be an emittance event which is either a method call from another actor to a or a return of a message sent by a to another actor. None of these possibilities fulfills the constraints for the output and internal signatures (Constraints A4 and A5). Thus, $e \notin out(\mathcal{A})$ and $e \notin int(\mathcal{A})$.

Now assume $e \in out(\mathcal{A})$. Constraint A4 states that e cannot be a reaction event. For e to also be in $int(\mathcal{A})$, it must be a self call (constraint A5). However, it violates constraint A4. Thus, $e \notin int(\mathcal{A})$ and the three signatures are pairwise disjoint. □

The lemma above reflects the signature stability of the class-based approach that provides the basis of our actor model.

Lemma 7.3 (Disjoint signatures between actor automata):
Let $\mathcal{A}$ and $\mathcal{A}'$ be actor automata of actors a and a', respectively, where $a \neq a'$. Then $in(\mathcal{A}) \cap in(\mathcal{A}') = out(\mathcal{A}) \cap out(\mathcal{A}') = int(\mathcal{A}) \cap int(\mathcal{A}') = \emptyset$.

Proof:
Assume that $e \in in(\mathcal{A})$. By constraint A3, e must either be a return emittance event with the future generated by a or a method call emittance event targeted to a. Because we assume $a \neq a'$, then by the same constraint $e \notin in(\mathcal{A}')$. Similar arguments are placed for the output signatures and internal signatures, applying the appropriate constraints. □

The following lemma shows that given AA that model two distinct actors, they are compatible. The lemma is framed under the condition that each actor only can send messages to the other actor that are in the set of allowed messages of the other actor (i.e., part of the (input) interface of the other actor). Employing type systems as done in ABS avoids such a problem.

Lemma 7.4 (Compatibility of actor automata):
Let $\mathcal{A}_1(a_1)$ and $\mathcal{A}_2(a_2)$ be AA representing any two distinct actors a_1 and a_2, respectively, such that $aMsg(class(a_1)) \cap \{e \mid target(e) = a_2\} \subseteq aMsg(class(a_2))$ and $aMsg(class(a_2)) \cap \{e \mid target(e) = a_1\} \subseteq aMsg(class(a_1))$. Then, $\mathcal{A}_1(a_1)$ is compatible with $\mathcal{A}_2(a_2)$.

Proof:
We need to establish that the compatibility of the signatures of the actor automata (Definition 6.4) are fulfilled. Let $s_1 \in states(\mathcal{A}_1(a_1))$ and $s_2 \in states(\mathcal{A}_2(a_2))$. It follows from the assumption and Constraints A3 to A5 that the internal state signature of s_2 cannot have a common intersection with the state signature of s_1 because the difference between future generator and the target actor of each event in either signature. The property of disjoint output state signatures is fulfilled following Lemma 7.3. Therefore the signatures of $\mathcal{A}_1(a_1)$ and $\mathcal{A}_2(a_2)$ are compatible and $\mathcal{A}(a_1)$ is compatible with $\mathcal{A}_2(a_2)$. □

The following property shows that the traces generated by an actor are stored properly in the state of the AA.

Proposition 7.2 (Generated traces of actors):
Let $\mathcal{A}(a)$ be an AA representing an actor a and $\alpha \in execs(\mathcal{A}(a))$ an execution that ends in state s. Then $trace(\alpha)\downarrow Gen(a) = s(t_{gen})$.

Proof:
The claim holds by induction on the length of the execution and Constraints A7 and A8. □

7.3 Discussion

This chapter presents an adaptation of SIOA to model the behavior of an actor. This adaptation features the use of non-shared futures. To model actors without futures, the state variables related to futures can be removed along with the relevant constraints. The structure of the constraints should remain the same.

Actor exposure plays an important role in verification, especially in the open system context. We can encode the information of which actors an actor a has exposed to some other actor b in the states of the AA. The advantage of including this information is that we can locally obtain the information whether a is exposed to the environment by means of composing the AA of a, the creator of a, say c, and relevant actors of the systems to which c exposes a to. A function can be created over a composed SIOA that aggregates the exposure mapping to obtain the desired information. We opt to regulate the actor exposure information to the configuration automaton level as presented in the next chapter. Apart from reducing the complexity of AA, the exposure information is needed to define the intrinsic signatures of the configurations.

Each state of AA stores the trace generated by the represented actor. This information is actually enough to infer the set of outgoing calls *outCalls*, future generator *futGen* and actor name generator *nameGen*. Its real value lies in the verification domain, where it eases the task of checking whether the class implementation satisfies the desired properties at release points (see Chapter 10).

Left out from the presentation of the DIOA adaptation for actor systems is the notion of fairness. The actor model stipulates that actors are fair, in the sense that the delivery of a message cannot be delayed indefinitely. We somewhat bypass this condition because in the 4-event semantics actors synchronize directly on emittance events. Further fairness conditions can be expected. For example, each actor that is currently not blocked has equal opportunity to progress. Similarly, given if an actor can infinitely often react to a certain message in its buffer, it should do so eventually. As with I/O automata [Lyn96, Section 8.3], these desired fairness conditions can be defined separately as a restriction on the traces.

CHAPTER 8

Configuration Automata for Actor Systems

In this chapter, we deal with how to refine the definition of CA to allow a well-behaved representation of an actor system. There are a few issues that need to be resolved when adapting CA for actor systems:

1. What information needs to be present in the configuration?

2. How should the signatures of the configurations be derived? Should the environment be given unconditional liberty to send input events to any actor in the system?

The answer to the first question is alluded in Chapter 6: the configuration should contain the AA that have been created, the state mapping, and the actors of the system exposed to the environment. The state of an AA representing actor a includes the information of the actors a knows, including other actors. However, AA are designed to be open and not to keep track what other actors do with information transmitted by the actor represented by the AA. To limit the interaction with the environment with actual possible interaction, the configuration contains explicitly the set of actors that have been exposed to the environment. We call this extended configurations *actor configurations*.

The intrinsic signatures of the actor configurations are defined following the observation that at most two actors are actively participating in an event. Whenever the two participating actors are part of the system, i.e., represented in an actor configuration, the events between them should be classified as internal because they are not observed by the environment. Because each AA places an actor directly in an open setting, there can be bogus transitions involving events either between two actors of the system or an actor of the system and an actor of the environment, but actually only one can participate in the transition. In general, this case appears when an actor is not yet exposed to another actor. The more subtle degenerate cases appear from the use of futures and the actor exposure.

The first subtle case appears when an actor a is exposed to another actor b and both are part of the system. The input signature of the AA representing a contains all input events that a can possibly receive from an (unknown from a's persepctive) b. However, the behavior of b is known, and this means that it is possible that b may not call certain methods of a. Even if b randomly calls all methods of a, some of the futures that b can generate have been previously generated, causing some input events not to have matching output events in the signature of the AA representing b. For a concrete example, consider the input signature of worker w after its exposure to server s. From the specification (Figure 7.1), s calls the `do` method of w only once, giving out only one of its generated futures to w. However, the input signature of w also contains other similar calls. These calls can only be executed by the AA representing w, but not the AA representing s.

Another issue that is already mentioned in Chapter 7, AA are not defined to control what information the environment can send to an (exposed) actor. For example, some actor a_e of the environment sends to an actor a_s of the system with a message whose acquaintance contains the actor name b. However if b is an actor of the system that is not yet exposed, or even worse, whose creator is a_s according to the *creator* relation, but it is actually not yet created, such an event is bogus. Because the exposure of an actor may be caused by other actors in the system, restricting such an input event is only done on the configuration level.

A similar argument holds as well for the futures generated by the environment. AA are defined to eliminate a future u being used twice to call an actor. However, an AA on its own does not eliminate the possibility of the environment from using u to call other actors. By storing these futures in the configuration, we ensure that a future uniquely identifies a method call.

To solve these problems, we throw such events out from the derived signatures. Therefore, we guarantee that all events in a transition are indeed events that happen when the actors interact.

Chapter outline. In the following section, we define the actor configurations, the intrinsic signatures of these configurations, the intrinsic transitions and the actor configuration automata (ACA) which represent open actor systems. Section 8.2 describes the properties of ACA. Finally, Section 8.3 relates the adaptation of DIOA described here to how actors can be represented in other kinds of automata, particularly those with a similar dynamic flavor.

8.1 Model

Definition 8.1 (Actor configuration and compatible actor configuration): An *actor configuration* $\mathbb{C}$ is a pair $\langle \mathbb{A}, \mathbb{S}, \mathbb{E}, \mathbb{U} \rangle$ where

- $\mathbb{A}$ is a set of AA (identifiers),
- $\mathbb{S}$ maps each AA (identifier) $\mathcal{A} \in \mathbb{A}$ to a state $s \in \mathit{states}(\mathcal{A})$,
- $\mathbb{E}$ is the set of actors (names) that have been exposed to the environment, and
- $\mathbb{U}$ is the set of futures that the environment has used in relation to calling methods of actors represented in the configuration.

We lift the *names* function to actor configurations, such that $\mathit{names}(\mathbb{C}) = \bigcup_{\mathcal{A} \in \mathbb{A}} \mathit{names}(\mathcal{A})$. An actor configuration $\mathbb{C}$ is *compatible* iff, for all $\mathcal{A}, \mathcal{B} \in \mathbb{A}$, $\mathcal{A} \neq \mathcal{B}$:

- $\mathit{names}(\mathcal{A}) \neq \mathit{names}(\mathcal{B})$, letting $a \in \mathit{names}(\mathcal{A})$ and $b \in \mathit{names}(\mathcal{B})$,
- $\mathit{out}(\mathcal{A})(\mathbb{S}(\mathcal{A})) \cap \{e \mid \mathit{isCreate}(e) \wedge \mathit{target}(e) = b\} = \emptyset$;

$\mathbb{E} \subseteq \mathit{names}(\langle \mathbb{A}, \mathbb{S}, \mathbb{E}, \mathbb{U} \rangle)$, and $\mathbb{U} \subseteq \{u \mid \mathit{ancestors}(\mathit{gen}(u)) \cap \mathit{names}(\mathbb{C}) = \emptyset\}$.

An actor configuration has four parts: the SIOA in the system, the state mapping, the set of actors that have been exposed to the environment and the set of futures the environment has used to call an actor of the system. The first two parts are similar to the configurations for CA (Definition 6.6), except that each SIOA is an actor automaton. The information on exposed actors and used futures allows an accurate derivation of intrinsic input signatures of the configuration.

The actor configuration reflects the state of an actor system. Therefore, the compatibility notion has to be adapted to take the actor setting into account. The first condition states that each SIOA in the actor configuration represents a unique actor. By Lemma 7.3, we know that a pair of actor automata have disjoint signature components, fulfilling the compatibility requirement of configurations as in Definition 6.6. So we only need to add the condition that no actor may produce a creation event that creates an actor that is already in the actor configuration. Finally, the exposed actors must be represented within the actor configuration and the used futures are generated by the environment. We continue to write $(\mathcal{A}, s) \in \mathbb{C}$ as a shorthand for $\mathcal{A} \in \mathbb{A} \wedge s = \mathbb{S}(\mathcal{A})$. The short hand $\mathit{exposed}(\mathbb{C})$

retrieves the actors in the configuration $\mathbb{C}$ that are exposed to the environment, while $\mathit{futs}(\mathbb{C})$ retrieves the set of futures the environment has used.

Example 8.1.1:
Let $\mathbb{C}$ be an actor configuration representing a `Server` actor s and a `Worker` actor w — $\langle\{\texttt{Server}(s), \texttt{Worker}(w)\}, \mathbb{S}, \{s\}, \{u\}\rangle$ — such that the server is in the state where it is processing a request task and just finished creating the worker. Only s is exposed to the environment (i.e., to the clients). It does not have other requests and the request task it is currently processing is the first request it receives. The worker has not received any input. This actor configuration is compatible because

- the two AA represent two different actors,
- the server can send method calls to the worker which are captured by the input interface of the worker (the specifications of the AA `Server` and `Worker` in Figures 7.1 and 7.2 and Constraint A8 of Definition 7.2),
- the worker does not send any calls to the server, trivially fulfilling the second condition from the worker side,
- the server cannot create another worker with the same name (Constraint A4 of Definition 7.2),
- the server is represented within the configuration, and
- the future is generated by some actor (i.e., a client) belonging to the environment. △

The intrinsic signatures of an actor configuration take into account the aforementioned subtleties. First we define the actor-external events of the actors represented in the configuration that represent events between two actors in the configuration. The common events are used to filter the union of input and output state signatures of individual SIOA in the configuration. Those that are present in both input and output state signatures of the SIOA in the configuration are inserted in the intrinsic internal signature. This filters out input events that are communicated between two actors of the configuration but are never actually generated while helping to preserve the input-enabledness requirement of CA. These common events include not only method calls and method returns, but also creation events because they should not be observed by the environment (Section 4.2).

The intrinsic external signature of the configuration reflects the possible message exchange between the system and its environment. The intrinsic output signature is completely controlled by the actors of the configuration. As these actors are well-behaved (Lemma 7.1), the focus is put on the intrinsic input signature.

The DIOA infrastructure requires all automata to be input-enabled. Thus, the input events from the environment must not include events where the environment uses actors that are supposed to belong to the system but are not yet exposed

to the environment and events the futures of which are in the set of used futures. To filter out such events, we identify the actors of the environment using the following function:

$$envActors(A) = \{a \mid ancestors(a) \cap A = \emptyset\}$$

where A is populated with the actors present in the system.

Definition 8.2 (Intrinsic signatures of an actor configuration):
Let $\mathbb{C} = \langle \mathbb{A}, \mathbb{S}, \mathbb{E}, \mathbb{U} \rangle$ be a compatible actor configuration. Let *commonEv* be the set of common events between actors represented within the configuration:

$$commonEv = \left\{ e \;\middle|\; \begin{array}{l} gen(e) \in names(\mathbb{C}) \wedge \\ (isMethod(e) \implies caller(e) \in names(\mathbb{C})) \wedge \\ (isCreate(e) \implies caller(e) \in names(\mathbb{C})) \end{array} \right\}.$$

Let *envEv* be the set of bogus events generated by the environment:

$$envEv = \left\{ e \;\middle|\; \begin{array}{l} isMethod(e) \wedge \\ (acq(e) \cap (envActors(names(\mathbb{C})) \cup \mathbb{E}) \neq \emptyset \vee fut(e) \in \mathbb{U}) \wedge \\ (isCall(e) \implies caller(e) \notin names(\mathbb{C})) \wedge \\ (isRet(e) \implies target(e) \notin names(\mathbb{C})) \end{array} \right\}.$$

Then, the signature $sig(\mathbb{C}) = \langle in(\mathbb{C}), out(\mathbb{C}), int(\mathbb{C}) \rangle$ is the *intrinsic signature* of $\mathbb{C}$, where

- $in(\mathbb{C}) = (\bigcup_{\mathcal{A} \in \mathbb{A}} in(\mathcal{A})(\mathbb{S}(\mathcal{A}))) - commonEv - envEv$,
- $out(\mathbb{C}) = (\bigcup_{\mathcal{A} \in \mathbb{A}} out(\mathcal{A})(\mathbb{S}(\mathcal{A}))) - commonEv$,
- $int(\mathbb{C}) = \bigcup_{\mathcal{A} \in \mathbb{A}} (int(\mathcal{A})(\mathbb{S}(\mathcal{A}))) \cup (\bigcup_{\mathcal{A} \in \mathbb{A}} in(\mathcal{A})(\mathbb{S}(\mathcal{A})) \cap \bigcup_{\mathcal{A} \in \mathbb{A}} out(\mathcal{A})(\mathbb{S}(\mathcal{A})))$,

The external signature $\mathbb{C}$ is defined as $ext(\mathbb{C}) = \langle in(\mathbb{C}), out(\mathbb{C}) \rangle$.

Example 8.1.2:
Following from the previous example, the intrinsic input signature of the configuration consists of all requests the server can receive from the environment (except for requests events with the same future attached to the currently processed request) and all `do` calls the worker can receive from the environment. The intrinsic output signature consists of new worker creation events by the server and the worker. The intrinsic internal signature consists of the requests that the server can pass on to the worker.

The intrinsic transitions are derived from transitions present in AA. The restriction placed on the intrinsic signatures characterizes how an actor configuration is affected when a transition is taken. Because the intrinsic signatures have eliminated events that could not take place, the following definition modifies Definition 6.8 by managing the exposed actors and the initial state of a created actor.

Definition 8.3 (Actor intrinsic transitions):
Let $\mathbb{C} = \langle \mathbb{A}, \mathbb{S}, \mathbb{E}, \mathbb{U} \rangle$, $\mathbb{C}' = \langle \mathbb{A}', \mathbb{S}', \mathbb{E}', \mathbb{U}' \rangle$ be arbitrary compatible actor configurations and e an event. Let $\mathcal{A}'(target(e))$ be an AA of class $class(e)$, if e is a creation event (i.e., $isCreate(e)$). There is an *actor intrinsic transition* from $\mathbb{C}$ to $\mathbb{C}'$ labeled by e, written $\mathbb{C} \overset{e}{\Rightarrow} \mathbb{C}'$, iff

1. $e \in \widehat{sig}(\langle \mathbb{A}, \mathbb{S}, \mathbb{E}, \mathbb{U} \rangle)$,
2. $\mathbb{A}' = \mathbb{A} \cup \{\mathcal{A}'(target(e)) \mid isCreate(e)\}$,
3. for all $\mathcal{A} \in \mathbb{A}' - \mathbb{A} : \mathbb{S}'(\mathcal{A}) \in start(\mathcal{A}) \wedge \mathbb{S}'(\mathcal{A})(params) = param(e)$,
4. for all $\mathcal{A} \in \mathbb{A}$: if $e \in \widehat{sig}(\mathcal{A})(\mathbb{S}(\mathcal{A})) \wedge \mathbb{S}(\mathcal{A}) \xrightarrow{e}_{\mathcal{A}} s$, then $\mathbb{S}'(\mathcal{A}) = s$, otherwise $\mathbb{S}'(\mathcal{A}) = \mathbb{S}(\mathcal{A})$,
5. $\mathbb{E}' = \mathbb{E} \cup \left\{ a \;\middle|\; \begin{array}{l} (isCall(e) \implies target(e) \notin names(\mathbb{C})) \wedge \\ (isRet(e) \implies caller(e) \notin names(\mathbb{C})) \wedge \\ a \in acq(e) - envActors(names(\mathbb{C})) \end{array} \right\}$, and
6. $\mathbb{U}' = \mathbb{U} \cup \{fut(e) \mid isCall(e) \wedge caller(e) \notin names(\mathbb{C})\}$.

Each actor intrinsic transition ensures that the event e is part of the intrinsic signature of the actor configuration. Different from the intrinsic transition, we know that at most one actor can be created as a result of executing an event. Moreover, the actor name is also exclusively derived from the event, which is in turn dependent on the state, as required by the definition of AA (Constraint A8 of Definition 7.2). As only compatible actor configurations are considered, the created actor is not yet represented by any SIOA in the configuration. The state of this new automaton is an initial state of the actor automaton with the class fields assigned to initial values as given in the parameter of the event. As in Definition 6.8, each SIOA that synchronizes on e makes a transition. In particular, the compatibility requirement ensures that whenever an input event of an actor in the configuration comes from another actor in the configuration, this event is synchronized with the output event of that actor. Event e that is sent to the environment may expose actors of the system. In such event the exposed actors

are recorded in the resulting configuration. The futures of method call events generated by the environment are recorded as being used.

Example 8.1.3:
Consider the same scenario as in the previous example with one server that is currently processing a request and has no other input, but the worker is only about to be created. The actor configuration $\mathbb{C} = \langle\{\texttt{Server}(s)\}, \mathbb{S}, \{s\}, \{u\}\rangle$ where

- $\mathbb{S}(\texttt{Server}(s))(\mathit{tasks}) = \{u \to s : \texttt{serve}\}$ and
- $\mathbb{S}(\texttt{Server}(s))(\mathit{buf}) = \mathbb{S}(\texttt{Server}(s))(\mathit{known}) = \emptyset$

reflects this scenario. The actor intrinsic transition $\mathbb{C} \xrightarrow{s \to w:\texttt{new Worker}()} \mathbb{C}'$ states the step of the server creating a worker actor, where $\mathbb{C}' = \langle\{\texttt{Server}(s), \texttt{Worker}(w)\}, \mathbb{S}', \{s\}, \{u\}\rangle$ such that $\mathbb{S}'(\texttt{Server}(s))(\mathit{known}) = \{w\}$ and for $s' = \mathbb{S}'(\texttt{Worker}(w))$, $s'(\mathit{buf}_c) = s'(\mathit{buf}_r) = s'(\mathit{known}) = s'(\mathit{tasks}) = s'(\underline{\mathsf{futTriple}}) = \emptyset$ (that is the `Worker` w is in its initial state).

The ACA, our extension of CA for actors, are defined as that for CA (Definition 6.9) except for the use of actor configurations and events. For simplicity, we restrict ourselves to ACA where initially there is only one actor in the actor configuration. Initial actor configurations that contain more than one actor can be simulated by having a main actor that creates these actors and send a start message to each of these actors in a non-deterministic order. Because the *created* mapping fully depends on the events, we do not need to specify this mapping as part of the definition.

Definition 8.4 (Actor configuration automata):
An *actor configuration automaton* $\mathcal{C}$ is a pair $\langle \mathit{sioa}(\mathcal{C}), \mathit{config}(\mathcal{C})\rangle$ where

- $\mathit{sioa}(\mathcal{C})$ is an SIOA;
  (As with CA, the parts of this SIOA are abbreviated to
  $\mathit{states}(\mathcal{C}) = \mathit{states}(\mathit{sioa}(\mathcal{C}))$, $\mathit{start}(\mathcal{C}) = \mathit{start}(\mathit{sioa}(\mathcal{C}))$, etc. for brevity)
- a configuration mapping $\mathit{config}(\mathcal{C})$ with domain $\mathit{states}(\mathcal{C})$ such that for all $x \in \mathit{states}(\mathcal{C})$, $\mathit{config}(\mathcal{C})(x)$ is a compatible actor configuration;

such that the following constraints are satisfied:

1. If $x \in \mathit{start}(\mathcal{C})$ and $(\mathcal{A}, s) \in \mathit{config}(\mathcal{C})(x)$, then $s \in \mathit{start}(\mathcal{A})$.
   Additionally, $\forall x \in \mathit{start}(\mathcal{C}) : \langle \mathbb{A}, \mathbb{S}, \mathbb{E}, \mathbb{U}\rangle = \mathit{config}(\mathcal{C})(x) \wedge |\mathbb{A}| = 1 \wedge \mathbb{E} \subseteq \mathit{names}(\mathbb{A}) \wedge \mathbb{U} = \emptyset$.
2. If $(x, e, x') \in \mathit{steps}(\mathcal{C})$ then $\mathit{config}(\mathcal{C})(x) \stackrel{e}{\Rightarrow} \mathit{config}(\mathcal{C})(x')$.

Definition 8.4 (Continued)

3. If $x \in states(\mathcal{C})$ and $config(\mathcal{C})(x) \overset{e}{\Rightarrow} \mathbb{C}$ for some event e and a compatible actor configuration $\mathbb{C}$, then $\exists x' \in states(\mathcal{C})$ such that $config(\mathcal{C})(x') = \mathbb{C}$ and $(x, e, x') \in steps(\mathcal{C})$.
4. For all $x \in states(\mathcal{C})$
   (a) $in(\mathcal{C})(x) = in(config(\mathcal{C})(x))$
   (b) $out(\mathcal{C})(x) = out(config(\mathcal{C})(x))$
   (c) $int(\mathcal{C})(x) = int(config(\mathcal{C})(x))$

Given an initial state x of ACA $\mathcal{C}$, we represent the initial actor of the configuration $\mathbb{C} = config(\mathcal{C})(x)$ by $init(\mathbb{C})$. Note that we do not fix the set of exposed actors in the initial state to be the initial actor. A *closed* system can be represented by having the set of exposed actors to be empty and the set of known actors in the state of the initial actor to contain only the initial actor. In such case, the creator of the initial actor is represented by $\perp$ to denote that the initial actor is by default present without any creator. The actors of the system represented by an ACA can be identified by checking whether their ancestors include the initial actor. The *env* predicate represents this check: given an actor a and an initial actor *init* of a system represented by an ACA,

$$env(a, init) = init \notin ancestors(a) \ .$$

The notions of executions and traces of an ACA remain the same as that of a CA. That is, we can reuse Definition 6.10.

The definition above allows the environment to use a future multiple times when calling exposed actors of the system. However, the environment can only use a future once to call a specific exposed actor of the system. Because future identities and actor names are unique, the generated events (and event cores) are distinguishable. If a more precise characterization is required, we can encode the received future as a specific SIOA and compose this SIOA with the control SIOA of the ACA. Alternatively, we can also define a CA for the environment such that futures are only used once and this CA is composed with the ACA. These options are out of scope for this thesis.

8.2 Properties

As ACA are the primary semantic objects representing actor systems, we have to ensure that ACA are well-behaved. This means, the traces of ACA should be well-

formed as stated by the following lemma. The proof idea for the lemma is similar to the lemma for AA, except that we consider actor configurations and intrinsic transitions.

Lemma 8.1 (Well-formedness of traces of ACA):
Let $\mathcal{C}$ be an ACA and $\alpha \in execs(\mathcal{C}) = x_0 \ldots$ is one of its non-empty executions such that *init* is the initial actor present in the execution (i.e., $init = init(config(\mathcal{C})(x_0))$). Then, execution α's trace $t = trace(\alpha)$ is well-formed with respect to the set of actors of the system $A = \{a \mid \neg env(a, init)\}$.

Proof:
All properties follow directly from the well-formedness of AA (Lemma 7.1) and how the signatures and transitions of actor configurations are derived (Definitions 8.2 and 8.3). □

We can extend the well-formedness definition to ensure that the environment does not use an actor of the system that is not yet exposed and show that the traces of ACA hold this property. Because the environment acts as a unit, any actor exposed to an actor of the environment is considered exposed to all actors of the environment, and renders them usable by all actors of the environment in their generated method calls or returns.

Definition 8.5 (Well-formed traces w.r.t. environment):
Let *init* be the initial actor of some actor system, and $A_{env} = \{a \mid env(a, init)\}$ be the set of actors of the system's environment. A trace t is *well-formed with respect to the environment* A_{env} if
$\forall t' \cdot e \in Pref(t{\downarrow}A_{env}) : isEmit(e) \wedge e$ is generated by $a \in A_{env} \implies acq(e) \subseteq acq(t') \cup A_{env}$

The following lemma shows that the environment as portrayed by the ACA is well-behaved.

Lemma 8.2 (Environment well-formedness of traces of ACA):
Let $\mathcal{C}$ be an ACA and $\alpha \in execs(\mathcal{C}) = x_0 \ldots$ is one of its non-empty executions such that *init* is the initial actor present in the execution (i.e., $init = init(config(\mathcal{C})(x_0))$). Let $A_{env} = \{a \mid env(a, init)\}$. Then, the execution α's trace $t = trace(\alpha)$ is well-formed with respect to A_{env}.

Proof:
Follows from Definition 8.4.2, Definition 8.3.5 and Definition 8.2. □

The disjointness of each signature part of an ACA state also follows.

Lemma 8.3 (Signature disjointness of ACA):
Let $\mathcal{C}$ be an actor configuration automaton. Then for all states $x \in states(\mathcal{C})$,

$$in(\mathcal{C})(x) \cap out(\mathcal{C})(x) = in(\mathcal{C})(x) \cap int(\mathcal{C})(x) = out(\mathcal{C})(x) \cap int(\mathcal{C})(x) = \emptyset \ .$$

Proof:
Follows from Definition 8.2 and Lemma 7.2. □

8.3 Discussion

In this section, we look into several automaton models. Section 8.3.1 describes automaton models that support dynamic creation and dynamic topology other than DIOA. Section 8.3.2 looks into automaton models that are specifically developed to model actors/objects, but do not address the dynamic creation issue. Discussion on other formal models is postponed to Part III.

8.3.1 Dynamic Automaton Models

The idea of representing a system featuring dynamic creation as a two-tier model is prevalent in the few automata-based models present in the literature. The automata are used to represent the individual entities, whereas other models may be used to represent the semantical model of systems. To our best knowledge, here are the automaton models that feature dynamic creation:

- Dynamic communicating automata [BH10; BCHKS13] extend communication automata ([BZ83]) to model the behavior of a process.
- Dynamic register automata [AAKR14] extend register automata ([KF94]) to model dynamic creation of processes, where each process is equipped with a number of registers.
- Callable timed automata [BVBF13] represent behavior templates for processes and the (timed) systems are represented using timed transition systems.
- Dynamic reactive module framework [Fis+11] represents process classes as simple dynamic discrete systems and the systems are represented using dynamic discrete systems.
- Dynamic I/O automaton model [AL01; AL15], which is adopted in this thesis.

Apart from DIOA which is presented in Chapter 6, we discuss these models along with other related automaton models used to represent (parts of) actor or object-based systems.

Dynamic communicating automata and dynamic register automata. Dynamic communicating automata (DCA) [BH10; BCHKS13] and dynamic register automata (DRA) [AAKR14] are extensions of classical automaton models, namely communication automata [BZ83] and register automata [KF94]. These models are developed to model a system of processes featuring dynamic creation and dynamic topology. Both DCA and DRA are finite state machines equipped with a finite, fixed set of registers. A register stores the identity of some process. Transitions between states are labeled with messages that can indicate the creation of new processes, send and receive messages. The DRA model is a bit richer than the DCA model by allowing to reset the values of registers. To send a message from one process to another process, the acquaintance of the message (i.e., the target and parameter identities) must be known to the sender before the message is sent.

Much like the DIOA model, the DCA and the DRA models also consist of two layers. The DCA and the DRA act as a template (in fact, the only one in these models) of how a process behaves. The behavior of a system is represented by a transition relation between configurations. Similar to CA configurations, a configuration in the DCA and the DRA models consists of a set of alive processes, a state mapping from alive processes to their automaton state, and a register mapping from alive processes to the values of their registers. The configuration of a DCA also includes the notion of channels which enables asynchronous communication. Unlike DCA, processes in the DRA model communicate synchronously, rendering it to be more restrictive with respect to distributed systems. The behavior of the system is represented by the runs based from the transition relation. Bollig et al. [BCHKS13] also describe the semantics of the DCA model in terms of message sequence charts ([IT11]).

The minimalist view of the DCA and DRA models makes it suitable for studying the boundary of automatic verification of systems that support dynamic creation and dynamic topology. However, there is still a big gap between these models and the actual (abstracted) implementation model, such as the actor model. In the implementation, an actor may hold the set of actor names that it has come to know so far (e.g., a subject actor in the observer pattern [GHJV95]). Therefore, a finite, *fixed* set of registers is unsatisfactory to represent such an actor. The notion of data also needs to be added to the models, for them to have a closer relationship with the implementation. Furthermore, these models pack all possible behavior

of a process into one single automaton with a single initial state. The separation of processes into several classes and how to compose these classes are yet to be investigated, as with the open environment setting.

Callable timed automata. The callable timed automaton (CTA) model [BVBF13] is an extension of timed automata (TA) [AD90] that allows the representation of real-time systems featuring dynamic creation. In this model, a system is populated with processes and a call made by a process to another process is represented by a CTA (i.e., the call itself is a process). CTA are parameterized with some identity and the description of a CTA is a template behavior of its instances. A CTA includes special call and return actions. When a transition labeled with a call action is taken, an automaton is created to process that call. The result of executing the call is transmitted by executing a transition with the return action as the label. The caller and the callee automata synchronize in the time transition system semantics (similar to CA) on matching input and output actions. A translation to TA is present, allowing the reuse of TA proof tools to reason over CTA. To model actors, the notion of actor names and exposure need to be introduced to the CTA model possibly in a similar way to our proposition.

Dynamic reactive module framework. Fisher et al. [Fis+11] introduce the dynamic reactive modules (DRM) as a means to model dynamic reconfiguration and creation of processes using transition systems. The DRM are classes with specific behaviors that can be instantiated in the form of simple dynamic discrete systems (SDDS). The transition relation of an SDDS is modeled by logical formulas. Initially, only one SDDS is active. Composition can be done on both the SDDS and on the dynamic reactive modules by combining common variables of the states and composing the transition relation accordingly. The dynamic system is modeled by a *finite* set of SDDS, named dynamic discrete system (DDS), while the static system is modeled by a finite set of reactive modules, named dynamic reactive system (DRS). In DDS, the creation of a class instance is handled via the execution of a **new** command, which returns the reference to the newly created instance. The communication between class instances is done exclusively through externally visible shared variables. However, as DDS contains only a fixed finite set of SDDS, the model handles only a bounded number of instance creations. A notion of refinement is present on the DDS and DRM level, but it is not present for the DRS.

8.3.2 Other Automaton Models

Jaghoori and Chothia [JC10] present a timed automata semantics to analyze the behavior of Creol ([JOY06]) objects. While the focus is more on the real-time aspect, the automata are also capable of representing desired functional behaviors. The automaton of an actor is built compositionally from the parts of the class implementation, unlike our actor automata which are specified separately from the implementation. No composition operation is defined for the actor automata. Furthermore, dynamic creation and dynamic topology are not considered.

Sirjani et al. [SJBA06] defined a translation of Rebeca ([SJ11]) models to constraint automata [BSAR06]. The translation is done in two stages: from Rebeca to Reo ([Arb04]), a coordination language, and then from Reo to constraint automata. The translation does not consider dynamic creation. Thus, the participating actors are present from the start. The dynamic topology is handled by a sweeping method that forces messages on Reo connectors representing actors to synchronize only when the actor names match. While there are works on dynamic reconfigurations for Reo (e.g., [KGV13]), allowing dynamic topology to be addressed in more detail, these techniques are not yet applied for actor-based setting.

Rumpe and Klein [RK96] propose an automaton model for objects called Message Processing Automata (MPA). The definition of MPA is very general, without any specific features to deal with dynamic topology. The semantics of an MPA is captured as a function of an infinite stream of input messages to an infinite stream of output messages. The input-enabledness property is captured by the separation of input and output messages. The message processing automata are taken as a system model on top of which a refinement calculus is defined. No composition operator on the automata is given.

In some respect our class specifications resemble the semi-automaton model [RD12] used to model objects of idealized Algol classes. The semi-automaton model allows a transition being a sequence of events before a state change occurs. While this model allows a smaller transition system for an actor (and fits nicely as a model for our class specification technique), the composition of multiple actor models usually requires the introduction of intermediate states to capture interleaving. Dynamic creation and dynamic topology are not yet established with this approach.

CHAPTER 9

Component Automata

One of our goals is to provide a model for the components that can be integrated to the modeling framework and facilitate the verification of a component implementation. In terms of the DIOA model, the components can be represented by an adaptation of the SIOA. Accordingly, there must be an adaptation of CA where the configurations accept component instances. Because this adaptation becomes a platform to verify that the combination of subcomponent specifications (including the class specifications) through parallel compositions satisfies the component specifications, these configurations must be general enough to also include AA. The adapted SIOA must be designed such that it can be composed with AA, producing what we call *actor-based* SIOA.

In this chapter, we present *component automata* (CompA) as an adaptation of SIOA that represent component instances. Focusing on how to represent the black box behavior of a component, CompA abstract from unnecessary details, namely the internal events other than the reaction events. Because the setting disallows creating actors of the environment, all creation events are ultimately internal events as the definition of intrinsic signatures of ACA suggests. As with AA, we refine the state space of an SA such that a CompA can be specified as concise as possible.

We introduce component configuration automata (CCA) to accommodate components as part of systems. CCA are based on *component configurations* which take into account the component instances. An intrinsic transition between two component configurations distinguishes between actors and component instances, such that when a component instance is created, a creation event involving that component instance produces a configuration where the component instance is composed with the new created entity. Otherwise, the intrinsic transitions of CCA have the same characteristics as the intrinsic transitions of ACA.

Chapter outline. This chapter is partitioned into five sections. The first two (Sections 9.1 and 9.2) describe the model and some properties of the CompA. The following two (Sections 9.3 and 9.4) describe the model and some properties of the CCA. In particular, we show that CCA are indeed a generalization of ACA.

Some discussion is provided at the end of the chapter.

9.1 Model of Component Automata

A CompA models the behavior of a group of actors whose initial actor $a \in \mathbf{A}$ is of an activator class C. That is, this CompA represents a component instance of initial class C. As with AA, we can also use the activator class name to refer to the CompA and use the set of variables $\mathbf{V}$ to refer to specific parts of a state. Because the behavior of classes is fixed, we can also parameterize CompA by the name of the initial actor. Similar to AA, states in CompA store information about the exposure of other actors to the component instance and the generated futures. In addition, CompA abstracts from the created actors by storing only the actors of a component instance that are exposed to the environment. In other words, the creation events are banned from the signatures of CompA. The restrictions on CompA are similar to AA, but with the consideration that the exposure of component actors may open up an interface that is not part of the initial actor's interface. Apart from using the *env* function defined in Section 8.1, we also use *envFut* to indicate whether a future is local to the component instance or generated by the environment.

$$envFut(u, init) = init \notin ancestors(gen(u))$$

The constant $comp = [class(init)]$ represents the boxed activator class of the initial actor *init* and is used to retrieve the allowed message with respect to actors of the component instance. The actor name can be retrieved by using the function *names* that maps a CompA (identifier) to a set of actors that can be created by *init* (i.e., $names(\mathcal{A}(init)) = \{a \mid init \in ancestors(a)\}$). The function *fut* is overloaded to accept a parameter of a set of non-creation events E, to return the set of futures attached to some event $e \in E$.

State variables. As with AA, CompA are instances of SA with the states containing necessary information to model our components. The state of a CompA contains buffers buf_c and buf_r that store the incoming calls and the method returns, respectively. In the component context, the call buffer may contain not just the events targeted to the initial actor of the component instance, but also events targeted to other actors of the component instance that have been exposed. Similarly, the task set *tasks* combines the tasks of all actors of the component. However, the use of *tasks* in CompA is less intensive than in AA because it is only employed to control the output state signature and transitions with an output event as label. For component instances, we follow the "external actors" and "receptionist

actors" approach by Agha et al. [AMST97] where the knowledge of exposed actors of the environment and the component instance is split into *known* and *expActors*, respectively. For simplicity, the set of actor names represented by a configuration of a CompA is equivalent to *expActors*.

To represent the state signatures correctly, a CompA is equipped with state variables storing the set of received future and target actor pairs, the set of outgoing calls and the set of generated futures. When an actor of the component instance is called by the environment, the future and the target actor of the call event are stored in *rcvFutTgt*. The pair, instead of just the future, is stored to imitate the same well-formed behavior as exhibited by AA and CCA. If we assume that the environment always produces unique futures, the target actor information can be removed. As in AA, we store the class parameters *params* of the initial actor of the component, as they can influence how the component acts. The internal variables used by the component is categorically represented by variable *ints*.

Transition relation constraints. The constraints show the relations between each state element with respect to the signatures and the transition relation. We follow the presentation of AA in explaining the constraints. The first constraint is a refined version of the original first constraint of SIOA, required to show that an CompA is indeed an SIOA. It states that every transition happens by executing an event of the state signature.

Initial state constraint. Constraint C2 states the generic property of the initial states. In an initial state, all sets are initialized to empty set, except for the known and the exposed actor sets. The known actor set contains the knowledge of all actors of the component instance as an overapproximation of the internal behavior of the component instance. It may also contain actor names that are part of the class parameters. Thus, the component instance is free to use these actors as part of the output events. The exposed actor set contains the initial actor of the component instance.

Constraint C3 ensures that events in the input state signatures are emittance events generated by the environment following the input interface of the component. The environment can only send a method call to an exposed actor of the component instance and the future and target actor pair must not have occurred. A return must have a matching outgoing call. The acquaintance of the event which are part of the component instance can only contain the exposed actors.

Definition 9.1 (Component automata):
A parameterized SA $\mathcal{A}(this) = \langle states(\mathcal{A}), start(\mathcal{A}), sig(\mathcal{A}), steps(\mathcal{A})\rangle$ with the following description:

- $states(\mathcal{A})$ is a map with a fixed domain $V \subseteq \mathbf{V}$ denoting the variables stored by the component instance. V includes the following fixed variables: buf_c, buf_r, *tasks*, *known*, *expActors*, *rcvFutTgt*, *outCalls* and *genFut* representing a call event buffer (of type $2^{\mathbf{E}}$), a return event buffer ($2^{\mathbf{E}}$), the set of tasks ($eCore(2^{\mathbf{E}})$), the set of known actors ($2^{\mathbf{A}}$), the set of exposed actors ($2^{\mathbf{A}}$), the set of received future and target actor pairs ($2^{\mathbf{U}\times\mathbf{A}}$), the set of outgoing calls ($eCore(2^{\mathbf{E}})$) and the generated futures ($2^{\mathbf{U}}$), respectively. The class parameters of the initial actor is stored under *params*. Other variables are internal and grouped together under *ints*;
- a non-empty set of initial states $start(\mathcal{A}) \subseteq states(\mathcal{A})$;
- a signature mapping $sig(\mathcal{A})$ where for each state $s \in states(\mathcal{A})$, $sig(\mathcal{A})(s) = \langle in(\mathcal{A})(s), out(\mathcal{A})(s), int(\mathcal{A})(s)\rangle$, where $in(\mathcal{A})(s), out(\mathcal{A})(s), int(\mathcal{A})(s) \subseteq \mathbf{E}$;
- a transition relation $steps(\mathcal{A}) \subseteq states(\mathcal{A}) \times acts(\mathcal{A}) \times states(\mathcal{A})$

is a *component automaton* representing a component instance with the initial actor of name *this* when it satisfies the following constraints:

C1. $\forall (s, e, s') \in steps(\mathcal{A}) : e \in \widehat{sig}(\mathcal{A})(s)$

C2. $\forall s \in start(\mathcal{A}) : s(buf_c) = s(buf_r) = s(tasks) = s(genFut) = \emptyset \wedge$
$s(rcvFutTgt) = s(outCalls) = \emptyset \wedge s(expActors) = \{this\} \wedge$
$\{a \mid this \in ancestors(a)\} \subseteq s(known)$

C3. $\forall s \in states(\mathcal{A}) : in(\mathcal{A})(s) =$
$$\left\{ e \;\middle|\; \begin{array}{l} isEmit(e) \wedge msg(e) \in aMsg(comp, in) \wedge \\ acq(e) - envActors(\{this\}) \subseteq s(expActors) \wedge \\ (isCall(e) \implies target(e) \in s(expActors) \wedge env(caller(e), this) \wedge \\ \qquad \langle fut(e), target(e)\rangle \notin s(rcvFutTgt)) \wedge \\ (isRet(e) \implies eCore(e) \in s(outCalls)) \end{array} \right\}$$

Definition 9.1 (Continued)

C4. $\forall s \in states(\mathcal{A}) : out(\mathcal{A})(s) =$
$$\left\{ e \;\middle|\; \begin{array}{l} isEmit(e) \wedge msg(e) \in aMsg(comp, out) \wedge \neg isCreate(e) \wedge \\ acq(e) \cap \{a \mid env(a, this)\} \subseteq s(known) \wedge \\ (isCall(e) \implies fut(e) \notin s(genFut) \wedge \neg envFut(fut(e), this) \wedge \\ \qquad \neg env(caller(e), this) \wedge target(e) \in s(known)) \wedge \\ (isRet(e) \implies eCore(e) \in s(tasks)) \end{array} \right\}$$

C5. $\forall s \in states(\mathcal{A}) :$
$$int(\mathcal{A})(s) = \{e \mid isReact(e) \wedge emitOf(e) \in s(buf_c) \cup s(buf_r)\}$$

C6. $\forall s \in states(\mathcal{A}) : \forall e \in in(\mathcal{A})(s) : \exists s' \in states(\mathcal{A}) : (s, e, s') \in steps(\mathcal{A}) \wedge$
$$s' = s \left[\begin{array}{l} buf_c \mapsto s(buf_c) \cup \{e \mid isCall(e)\}, buf_r \mapsto s(buf_r) \cup \{e \mid isRet(e)\}, \\ rcvFutTgt \mapsto s(rcvFutTgt) \cup \{\langle fut(e), target(e)\rangle \mid isCall(e)\}, \\ outCalls \mapsto s(outCalls) - \{eCore(e) \mid isRet(e)\} \end{array} \right]$$

C7. $\forall (s, e, s') \in steps(\mathcal{A}) : \exists s'' : isReact(e) \implies diffOn(s, s'', \{ints\}) \wedge$
$$s' = s'' \left[\begin{array}{l} buf_c \mapsto s''(buf_c) - \{emitOf(e)\}, \\ known \mapsto s''(known) \cup (acq(e) - s''(expActors)), \\ tasks \mapsto s''(tasks) \cup \{eCore(e) \mid isCall(e)\} \end{array} \right]$$

C8. $\forall (s, e, s') \in steps(\mathcal{A}) : \exists s'' : isEmit(e) \wedge e \in out(\mathcal{A})(s) \implies$
$diffOn(s, s'', \{ints, expActors\}) \wedge$
$s''(expActors) = s(expActors) \cup (acq(e) - \{a \mid env(a, this)\}) \wedge$
$$s' = s'' \left[\begin{array}{l} genFut \mapsto s''(genFut) \cup \{fut(e) \mid isCall(e)\}, \\ tasks \mapsto s''(tasks) - \{eCore(e) \mid isRet(e)\}, \\ outCalls \mapsto s''(outCalls) \cup \{eCore(e) \mid isCall(e)\} \end{array} \right]$$

The output state signatures depend on the tasks of the component instance and exposed actors of the environment. Constraint C4 accumulates all method returns that the component can produce and all method calls it can make to the environment as the output signature of each state of CompA. Because it is not important on the component level to know which actor makes the call, the generator of the future is not restricted as long as it is an actor of the component instance. The message of the event must be part of the output interface of the component instance. The acquaintance of the event which is part of the environment can only contain the exposed actors of the environment.

Transition relation constraints. Constraints C6 to C8 deal with the restriction on the transition relation and the changes in state input signatures. Constraint C6

Allowed messages

`Server`():
provided: `Value serve(Query q)`
required: none

State

$\underline{\text{futQPair}} \subseteq \mathbf{U} \times \mathbf{D}(\texttt{Query})$, **initially** $\emptyset$

Actions

$u \twoheadrightarrow this : \texttt{serve}(q)$		$u \leftarrow this : \texttt{serve} \triangleleft \texttt{compute}(q)$	
pre:	true	**pre**:	$\langle u, q \rangle \in \underline{\text{futQPair}}$
state:	$\underline{\text{futQPair}} := \underline{\text{futQPair}} \cup \{\langle u, q \rangle\}$	**state**:	$\underline{\text{futQPair}} := \underline{\text{futQPair}} - \{\langle u, q \rangle\}$

Figure 9.1.: A specification of component automaton [`Server`](*this*) for [`Server`] component

deals exclusively with the input events, while Constraints C7 and C8 describe the effects of generating reaction and emittance events, respectively.

As with AA, the input-enabledness property (Constraint C6) ensures that all input events go into the buffer. Executing an input event places the event into the buffer. If the event is a method call, the pair of future and target actor is stored in the state. If the event is a method return, the corresponding outgoing call is removed from the state.

Constraint C7 states that a reaction event enriches the knowledge of the component instance about the environment. The corresponding emittance event is removed from the buffer, and if it is a method call, then a new task is added to the component. The internal state may be modified.

Executing an output emittance event may expose more actors of the component instance and change the internal state as governed by C8. If the event is a method call, a new future is generated and the outgoing call is stored. If the event is a method return, the corresponding task is removed from the set of tasks.

Example 9.1.1:
The CompA of `Server` and `Worker` as activator classes are specified in Figures 9.1 and 9.2. As with AA, the specification for CompA consists of three parts: the allowed messages, its internal state and its actions. Of these three parts, only the **State** section follows the same format. The **Allowed messages** section contains a set of classes of actors that may be exposed by the component instance. For each class, it specifies the incoming calls an exposed actor of that class can handle.

Allowed messages

`Worker`():
provided: `Value do(Query` q`)`
required: none

State

$\underline{\text{futQPair}} \subseteq \mathbf{U} \times \mathbf{D}(\texttt{Query})$, **initially** $\emptyset$

Actions

$u \twoheadrightarrow \mathit{this} : \texttt{do}(q)$	$u \leftarrow \mathit{this} : \texttt{do} \triangleleft \texttt{compute}(q)$
pre: true	**pre**: $\langle u, q \rangle \in \underline{\text{futQPair}}$
state: $\underline{\text{futQPair}} := \underline{\text{futQPair}} \cup \{\langle u, q \rangle\}$	**state**: $\underline{\text{futQPair}} := \underline{\text{futQPair}} - \{\langle u, q \rangle\}$

Figure 9.2.: A specification of component automaton [`Worker`](*this*) for [`Worker`] component

Component instances may call exposed environment actors. The types of the actors can be gathered from the parameters of the incoming calls and can be used for determining which calls the component can make depending on the exposed actors. Because CompA disallow internal events beyond the reaction events which can be derived from the input messages, there is no need to specify messages that are only used internally. This follows the focus of CompA on representing the externally observable behavior of component instances. The **Actions** section specifies the transitions a component instance may take. Unlike actors, component instances in general cannot provide a sequential guarantee when processing a message. For example, the computation of the requests can finish at differing times and the client can receive the result of the requests not in the order they are processed by the server. Therefore, the behavior of a component instance is defined per event, as with I/O automata [Lyn96, pp. 203–204].

Both the [`Server`] and the [`Worker`] component specifications are expected to produce the same behavior, barring the slight method name difference for the requests. Thus, it is sufficient to explain only one of them. The [`Server`] component specification consists of these three parts. Because the initial actor is the only exposed actor, we only need to specify the messages the initial actor can receive based on the class of the initial actor. As a result, the **provided** part of the `Server` class remains the same as in Example 7.1.1. The **required** part on the other hand becomes empty, because the component instances can never call an actor of the environment.

Unlike the internal state of the `Server` class specification, the internal state of [`Server`] component specification consists of a set of pairs of a future and a query. The information of the future generated by the server when delegating the query to a worker is abstracted. In other words, we do not get to see how each request is processed as shown by the **Actions** section.

The **Actions** section of the component specification illustrates only the start and end points of the [`Server`] component processing the request. The only control mechanism needed here is that the futures need to be resolved with the correct computation results.

In Section 10.2 we provide the semantics of the component specifications in terms of CompA. Similar to the semantics of the class specifications, we populate the states, signatures and transitions of CompA appropriately, such that the constraints placed on the CompA are fulfilled. From the [`Server`] specification, the corresponding CompA $\mathcal{A}(s)$ can produce the following trace for a component instance with the initial server actor s.

$$u \rightarrow s : \texttt{serve}(q) \cdot u \twoheadrightarrow s : \texttt{serve}(q) \cdot u \leftarrow s : \texttt{serve} \triangleleft \texttt{compute}(q)$$

△

9.2 Properties of Component Automata

In this section we look into several properties of CompA. The most fundamental one is that CompA are SIOA. Thus, we have uniform definitions of executions and trace.

Proposition 9.1:
A component automaton $\mathcal{A}$ is an SIOA.

Proof:
$\mathcal{A}$ is an SA, thus only the constraints of Definition 6.2 need to be checked. The first constraint follows directly from Constraint C1, whereas the second constraint on input-enabledness follows from Constraint C6. The disjointness of the input, output and internal state signatures as required by the third constraint follows directly from Constraint C3 to C5. □

We expect the traces of CompA to be well-formed (Definition 4.3). As with AA, the idea behind the proofs of these properties is that the constraints of CompA capture the well-formedness properties.

Lemma 9.1 (Well-formedness of traces of component automata):
Let $\mathcal{A}(a)$ be a component automaton with initial actor a. Let $\alpha \in execs(\mathcal{A}(a))$ be an execution of $\mathcal{A}(a)$ such that t is the derived trace of α. Then, t is well-formed with respect to $\{a' \mid a \in ancestors(a)\}$.

Proof:
The proof follows from the constraints on component automata.
Property 1: If there is no event in α with the future u, the property holds. Assume the projection of the trace to the future u results in events $t' = e_1 e_2 e_3 \ldots$. The first premise needs to be shown is that the first event of this projected trace t' is a method call emittance event. By Constraint C3, a method return event is present in the input signature, but only if the corresponding event core is in the set of tasks. Similar argument is fulfilled for the output signature by Constraint C4. Initially the set of tasks is empty. For an event core to be in the set of tasks, the corresponding input event has to be in the buffer (Constraints C7 and C8). Thus, the premise holds. We proceed by proving what happens after the method call emittance event is received.
Case *internal call*: This case never happens because no internal call is part of the signature, following Constraints C3 to C5. Thus, the property trivially holds.
Case *incoming method call*: Let e_1 be a method call emittance input event (i.e., $caller(e_1) \in envActors(\{a\})$). By Constraint C6, e_1 is placed into the buffer and the future of e_1 (and a) is stored in the set of received futures, effectively removing all events with the same future from the input signature (Constraint A3). When $|t'| > 1$, e_2 must be a reaction event of e_1, because no further input with the same future and actor pair may appear and no task with the same future and actor name pair is present in the set of tasks (Constraint C7). The corresponding event core of e_2 is inserted into the task set. As the input signature excludes method call events whose futures have been received previously, the automaton never returns back to the state with an input signature that contains e_1. When $|t'| > 2$, the only option is to finish processing the task and return the result of executing the method (Constraint C8). Then it follows that the projected trace may only contain a method call event followed by the corresponding method return event.
Case *outgoing method call*: Let event e_1 be a method call emittance output event, that is $caller(e_1) = a \wedge target(e_1) \neq a$. e_1 cannot be a reaction event as the only possibility that the future of e_1 is generated by a is that e_1 must be an output event. Following Constraint C8, such an event only appears in the output signature. Because the generated future is stored, the same future will never be generated again (Constraint C4). The input signature is updated accordingly to allow for the corresponding method return events. For the projected trace to pro-

ceed, only a corresponding method return event may appear as an input. That is, e_2 is of the form $u \leftarrow b.mtd \triangleleft v$ for some result v. Following Constraint C6, this input event is stored in the buffer. The only extension left for the projected trace is the reaction event of e_2 which can be done repeatedly. Thus, the property holds.

Property 2: Let α be such that $trace(\alpha) = t \cdot e$ where e is an emittance event and generated by a. For any transition (s, e', s') such that $s\ e'\ s'$ is a part of α, the set of known actors grows monotonically ($known(s) \subseteq known(s')$) following Constraints C6 to C8. As e appears in $trace(\alpha)$ and generated by a, a transition $(s_e, e, s'_e) \in steps(\mathcal{A})$ is taken. As Property 1 holds, e can only appear once in $trace(\alpha)$, meaning it cannot appear in t. For transition (s_e, e, s'_e) to be executable, Constraint C8 must be fulfilled, particularly $acq(e) \subseteq known(s_e)$. Similarly, every consecutive triple (s, e', s') in α must obey the Constraints C4 and C5, maintaining the monotonicity property. As t can contain input events that are not reacted to, $known(s_e) \subseteq acq(t) \cup \{a' \mid a \in ancestors(a')\}$. Thus, $acq(e) \subseteq acq(t) \cup \{a' \mid a \in ancestors(a')\}$.

Property 3: Follows directly from Constraints C8. □

The definition of CompA places more guarantee such that the generated traces of a CompA are also well-formed with respect to its environment (Definition 8.5).

Lemma 9.2 (Environment well-formedness of traces of CompA):
Let $\mathcal{A}$ be a component automaton with initial actor a and $\alpha \in execs(\mathcal{A})$ is an execution. Let $A_{env} = envActors(\{a\})$. Then the trace $t = trace(\alpha)$ is well-formed with respect to the environment $A_{env} = envActors(\{a\})$.

Proof:
Follows from Constraints C3 and C8. □

The constraints of CompA ensure that the state signature of a CompA only contains external emittance events and their reaction events.

Proposition 9.2 (State signature event characteristic of CompA):
Let $\mathcal{A}$ be a component automaton with initial actor a and

$$E_{cmp}(a) = \{e \mid isMethod(e) \wedge (caller(e) \notin ancestors(a) \vee target(e) \notin ancestors(a))\}$$

the set of external events and their corresponding reaction events with re-

Proposition 9.2 (Continued)
spect to a component instance with initial actor a. Then,

$$\forall s \in states(\mathcal{A}) : sig(\mathcal{A})(s) \subseteq E_{cmp}(a) \ .$$

Proof:
From Constraints C3 and C4, $ext(\mathcal{A})(s) \subseteq E_{cmp}(a)$. Since by Constraint C5 $int(\mathcal{A})(s)$ only adds reaction events to the emittance events in the buffer, which is only populated with input events, $sig(\mathcal{A})(s) \subseteq E_{cmp}(a)$. □

The disjointness of the signatures within a CompA and between CompA follows from the disjointness of the individual actor, that the internal events can only be reaction events to input events, and the fact that the exposed actors must have the initial actor of the component as one of their ancestors.

Proposition 9.3 (Disjoint component automaton signature):
Let $\mathcal{A}$ be a component automaton. Then,

$$in(\mathcal{A}) \cap out(\mathcal{A}) = in(\mathcal{A}) \cap int(\mathcal{A}) = out(\mathcal{A}) \cap int(\mathcal{A}) = \emptyset \ .$$

Proof:
From the Constraint C5 it is clear that internal events are exclusively reaction events and reaction events are never categorized as input or output events. Therefore, we only need to show that $in(\mathcal{A}) \cap out(\mathcal{A}) = \emptyset$. From Constraint C4, an output event is either a return event targeted to an actor of the component or a method call targeted to an exposed environment actor, while according to Constraints C3 and C8, an input event is the opposite. Therefore, the input and output signatures are also disjoint. □

Proposition 9.4 (Disjoint signatures between component automata):
Let $\mathcal{A}$ and $\mathcal{A}'$ be component automata with initial actors a and a', respectively, where $a \notin ancestors(a') \wedge a' \notin ancestors(a)$. Then, $in(\mathcal{A}) \cap in(\mathcal{A}') = out(\mathcal{A}) \cap out(\mathcal{A}') = int(\mathcal{A}) \cap int(\mathcal{A}') = \emptyset$.

Proof:
The lemma follows from the assumption that the actors of the components being disjoint. □

The signature disjointness between a CompA and an AA also follows if the actor represented by the AA is part of the environment of CompA.

Proposition 9.5 (Disjoint signatures between AA and CompA):
Let $\mathcal{A}$ be a component automaton with the initial actor a and $\mathcal{A}'$ be an actor automaton representing actor a' such that $a' \notin ancestors(a)$. Then, $in(\mathcal{A}) \cap in(\mathcal{A}') = out(\mathcal{A}) \cap out(\mathcal{A}') = int(\mathcal{A}) \cap int(\mathcal{A}') = \emptyset$.

Proof:
Follows from Constraints A3 to A5, Constraints C3 to C5 and the assumption that $a' \notin ancestors(a)$. □

The propositions above indicate that CompA, as with AA, fulfill the requirements of the parallel composition operator. We classify SIOA that are obtained from the composition of CompA and AA as actor-based SIOA. We use $s(a)$ to extract the state of a particular actor or a component instance with the initial actor a from the state s of an actor-based SIOA $\mathcal{A}$.

Definition 9.2 (Actor-based SIOA):
Let $\{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ be a set of AA and CompA. The SIOA $\mathcal{A} = \mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n$ is an *actor-based* SIOA.

9.3 Model of Component Configuration Automata

In this section, we realize an adaptation of CA, called *component configuration automata* (CCA), that accommodates components. To define CCA, first, we include the information of which classes are the activator classes. Second, we extend the configuration to include information which SIOA represents an actor and which SIOA represents a component instance. Every time there is a transition that creates an actor of an activator class C, the new SIOA (that is, an instance of AA of C) is marked as a component, unless the creator is part of a component instance. In the latter case, the new SIOA is composed together (using the parallel composition operator) with the creator's SIOA. Otherwise, this transition creates a new SIOA and puts it into the configuration as done in ACA. Transitions that do not involve actor creations are modeled the same to what is done in ACA with the exception of allowing simultaneous exposure of a group of actors. We show that this adaptation does not impact the observable behavior of the represented actor system in comparison to ACA.

To unite the use of AA and CompA, CCA are defined such that components $[C]$ can be designated as activator classes. Creating such a component means instantiating a parameterized component automaton that represents $[C]$. The

component automata are SIOA, allowing the AA, composed AA and component automata to be treated in a uniform manner.

First we define a notion of configurations called *component configuration* that allows the use of an SIOA $\mathcal{A}$ to represent a group of actors. To distinguish SIOA that represent only single actors from SIOA that represent a group of actors, we encode this information in a boolean mapping. The notion of a compatible component configuration is a generalization of the notion of compatible actor configurations.

Definition 9.3 (Component configurations):
A *component configuration* $\mathbb{C}$ is a tuple $\langle \mathbb{A}, \mathbb{S}, \mathbb{E}, \mathbb{U}, \mathbb{B} \rangle$ where

- $\mathbb{A}$ is a set of SIOA identifiers,
- $\mathbb{S}$ maps each SIOA identifier $\mathcal{A} \in \mathbb{A}$ to a state $s \in \mathit{states}(\mathcal{A})$,
- $\mathbb{E}$ is the set of actors (names) that have been exposed to the environment,
- $\mathbb{U}$ is the set of futures that the environment have used in relation to calling methods of actors represented in the configuration, and
- $\mathbb{B}$ maps each SIOA identifier $\mathcal{A} \in \mathbb{A}$ to a boolean value, indicating, if true, that $\mathcal{A}$ represents a component instance. Otherwise, $\mathcal{A}$ is an AA representing an actor.

We lift the *names* function to component configurations, such that $\mathit{names}(\mathbb{C}) = \bigcup_{\mathcal{A} \in \mathbb{A}} \mathit{names}(\mathcal{A})$. A component configuration $\mathbb{C} = \langle \mathbb{A}, \mathbb{S}, \mathbb{E}, \mathbb{U}, \mathbb{B} \rangle$ is *compatible* iff, for all $\mathcal{A}, \mathcal{B} \in \mathbb{A}$, $\mathcal{A} \neq \mathcal{B}$:

- $\mathit{names}(\mathcal{A}) \cap \mathit{names}(\mathcal{B}) = \emptyset$,
- $\mathit{out}(\mathcal{A})(\mathbb{S}(\mathcal{A})) \cap \{e \mid \mathit{target}(e) \in \mathit{names}(\mathcal{B}) \vee \mathit{caller}(e) \in \mathit{names}(\mathcal{B})\}$
  $\subseteq \mathit{in}(\mathcal{B})(\mathbb{S}(\mathcal{B})) \cap \{e \mid \mathit{caller}(e) \in \mathit{names}(\mathcal{A}) \vee \mathit{caller}(e) \in \mathit{names}(\mathcal{B})\}$, and
- $\mathit{out}(\mathcal{A})(\mathbb{S}(\mathcal{A})) \cap \{e \mid \mathit{isCreate}(e) \wedge \mathit{target}(e) \in \mathit{names}(\mathcal{B})\} = \emptyset$.

$\mathbb{E} \subseteq \mathit{names}(\mathbb{C})$ and $\mathbb{U} \subseteq \{u \mid \mathit{ancestors}(\mathit{gen}(u)) \cap \mathit{names}(\mathbb{A}) = \emptyset\}$.
Additionally, for all $\mathcal{A} \in \mathbb{A} : |\mathit{names}(\mathcal{A})| > 1 \implies \mathbb{B}(\mathcal{A}) = \mathit{true}$.

The definition above is a natural extension to the actor configuration definition (Definition 8.1), where it provides also a map to indicate which SIOA in the configuration are representing groups of actors. The compatibility of component configurations only differs from that of actor configurations in the assumption that each SIOA represents an actor. Apart from that, the compatibility notion remains

the same. An output event generated by an actor represented within some actor-based SIOA $\mathcal{A}$ and sent to an actor represented within $\mathcal{B}$ can be accepted as an input event for the target actor, and no creation event that marks the creation of an actor that is already in the actor configuration is generated. The compatibility notion above assumes that there is no conflict within an SIOA, which is guaranteed by how the transitions are intrinsically derived. This notion also strengthens the usage of the mapping, such that when an SIOA represents a component instance, it is marked so in the mapping $\mathbb{B}$, which we call *component instance mapping*. The function *names* for actor configurations retains the same meaning for component configurations. This function applied to a component automata instance returns the set of all actors that may be transitively created by the initial actor of the component instance. We qualify $(\mathcal{A}, s, b)$ as a member of a component configuration $\langle \mathbb{A}, \mathbb{S}, \mathbb{E}, \mathbb{U}, \mathbb{B} \rangle$ if $\mathcal{A} \in \mathbb{A}$, $\mathbb{S}(\mathcal{A}) = s$ and $\mathbb{B}(\mathcal{A}) = b$. The functions *exposed*$(\mathbb{C})$ and *futs*$(\mathbb{C})$ retain their meaning for actor configurations.

The notion of intrinsic signatures of component configurations is formalized by Definition 9.4. This notion is the same as that for actor configurations, because the additional element $\mathbb{B}$ does not affect the actual signatures of each SIOA within a configuration.

As with the intrinsic signatures, the intrinsic transitions for component configurations do not differ much from the intrinsic transitions for actor configurations. There are two main differences: the consideration of groups of actors as component instances, instead of just single actors, and the inclusion of activator classes to distinguish which groups of actors should be represented by an SIOA. Unlike for actor configurations where an actor creation event simply means adding a new SIOA to the set of existing SIOA, in the component case we have to distinguish whether the new SIOA is going to be composed with the SIOA that generates the event or it is going to be a separate member of the configuration. The latter case requires a further check on whether the class of created actor is an activator class. If so, the component instance mapping for the corresponding SIOA marks this SIOA as a component. Furthermore, if the activator class is represented in its boxed variant, the component automaton instance is used in the configuration instead of the actor automaton one. Because an SIOA may represent a group of actors, more than one actor of that group can be exposed within one single event.

A component intrinsic transition (Definition 9.5) relies on the event being in the intrinsic signature of the component configuration. If e is a creation event such that the creator is part of a component instance, then the SIOA representing the component instance is composed with the new SIOA in the configuration. If e is a creation event where the creator is not part of a component instance, the new SIOA is a new member of the configuration. Furthermore, if the class of

Definition 9.4 (Intrinsic signatures of a component configuration): Let $\mathbb{C} = \langle \mathbb{A}, \mathbb{S}, \mathbb{E}, \mathbb{U}, \mathbb{B} \rangle$ be a compatible component configuration. Let *commonEv* be the set of common events between actors represented within the configuration:

$$commonEv = \left\{ e \;\middle|\; \begin{array}{l} target(fut(e)) \in names(\mathbb{C}) \wedge \\ (isMethod(e) \implies caller(e) \in names(\mathbb{C})) \wedge \\ (isCreate(e) \implies caller(e) \in names(\mathbb{C})) \end{array} \right\}.$$

Let *envEv* be the set of bogus events generated by the environment:

$$envEv = \left\{ e \;\middle|\; \begin{array}{l} isMethod(e) \wedge \\ (acq(e) \cap (envActors(names(\mathbb{C})) \cup \mathbb{E}) \neq \emptyset \vee fut(e) \in \mathbb{U}) \wedge \\ (isCall(e) \implies caller(e) \notin names(\mathbb{C}) \wedge \\ (isRet(e) \implies target(e) \notin names(\mathbb{C})) \end{array} \right\}.$$

Then, the signature $sig(\mathbb{C}) = \langle in(\mathbb{C}), out(\mathbb{C}), int(\mathbb{C}) \rangle$ is the *intrinsic signature* of $\mathbb{C}$, where

- $in(\mathbb{C}) = (\bigcup_{\mathcal{A} \in \mathbb{A}} in(\mathcal{A})(\mathbb{S}(\mathcal{A}))) - commonEv - envEv$
- $out(\mathbb{C}) = (\bigcup_{\mathcal{A} \in \mathbb{A}} out(\mathcal{A})(\mathbb{S}(\mathcal{A}))) - commonEv$
- $int(\mathbb{C}) = (\bigcup_{\mathcal{A} \in \mathbb{A}} int(\mathcal{A})(\mathbb{S}(\mathcal{A}))) \cup (\bigcup_{\mathcal{A} \in \mathbb{A}} in(\mathcal{A})(\mathbb{S}(\mathcal{A})) \cap \bigcup_{\mathcal{A} \in \mathbb{A}} out(\mathcal{A})(\mathbb{S}(\mathcal{A})))$

The external signature $\mathbb{C}$ is defined as $ext(\mathbb{C}) = \langle in(\mathbb{C}), out(\mathbb{C}) \rangle$.

the created actor is an activator class, the new SIOA is marked as a component instance. The exposure of actors is done in bulks, as an SIOA may represent a group of actors. If an event is sent to the environment, all acquaintance of that event is exposed to the environment. The collection of futures used by the environment also remains the same. The transition causes the state of an SIOA in the configuration to change if an actor represented by the SIOA takes part in the event. No transition causes the mapping of component instances to change on existing SIOA.

The definition of CCA differs from ACA only on the use of component configurations and component intrinsic transitions. As explained previously, the activator class info must be included in the structure of CCA. To avoid confusion, an acti-

Definition 9.5 (Component intrinsic transitions):
Let $\mathbb{C} = \langle \mathbb{A}, \mathbb{S}, \mathbb{E}, \mathbb{U}, \mathbb{B} \rangle$, $\mathbb{C}' = \langle \mathbb{A}', \mathbb{S}', \mathbb{E}', \mathbb{U}', \mathbb{B}' \rangle$ be two arbitrary compatible component configurations and e an event. Let $CL \subseteq \mathbf{CL} \cup [\mathbf{CL}]$ be a set of classes and components. If e is a creation event and $class(e) \in CL$, we let $\mathcal{A}'' = \mathcal{A}'(target(e))$ be an actor automata of class $class(e)$. If e is a creation event and $[class(e)] \in CL$ is a boxed class $[C]$, we let $\mathcal{A}'' = \mathcal{A}'(target(e))$ be a component automata of initial class C. There is a *component intrinsic transition* from $\mathbb{C}$ to $\mathbb{C}'$, written as $\mathbb{C} \overset{e}{\Rightarrow}^{CL} \mathbb{C}'$, iff

1. $e \in \widehat{sig}(\mathbb{C})$;
2. Let $\mathbb{A}''$ be a set of SIOA identifiers such that,
   - if e is a creation event (i.e., $isCreate(e)$), $\mathcal{A} \in \mathbb{A}$ is an SIOA such that $caller(e) \in names(\mathcal{A})$ and $\mathcal{A}$ is derived from a group of actor automata (i.e., $\mathbb{B}(\mathcal{A}) = true$),

     $$\mathbb{A}'' = (\mathbb{A} - \{\mathcal{A}\}) \cup \{\mathcal{A} \parallel \mathcal{A}''\} ;$$

     in addition, $\mathbb{S}'(\mathcal{A} \parallel \mathcal{A}'') = \langle s', s'' \rangle$ where $\mathbb{S}(\mathcal{A}) \xrightarrow{e}_{\mathcal{A}} s'$ and $s'' \in start(\mathcal{A}'')$ such that $param(e) = s''(params)$;
   - if e is a creation event (i.e., $isCreate(e)$), $\mathcal{A} \in \mathbb{A}$ is an SIOA such that $caller(e) \in names(\mathcal{A})$ and $\mathcal{A}$ is an AA (i.e., $\mathbb{B}(\mathcal{A}) = false$),

     $$\mathbb{A}'' = \mathbb{A} \cup \{\mathcal{A}''\} ;$$

     in addition, $\mathbb{S}'(\mathcal{A}'') \in start(\mathcal{A}'')$ such that $param(e) = param(\mathbb{S}'(\mathcal{A}''))$, $\mathbb{B}'(\mathcal{A}'') = class(e) \in CL$;
   - otherwise $\mathbb{A}'' = \mathbb{A}$;
3. for all $\mathcal{A} \in \mathbb{A} : \mathbb{B}'(\mathcal{A}) = \mathbb{B}(\mathcal{A})$ and if $e \in \widehat{sig}(\mathcal{A})(\mathbb{S}(\mathcal{A})) \wedge \mathbb{S}(\mathcal{A}) \xrightarrow{e}_{\mathcal{A}} s$, then $\mathbb{S}'(\mathcal{A}) = s$, otherwise $\mathbb{S}'(\mathcal{A}) = \mathbb{S}(\mathcal{A})$,
4. $\mathbb{E}' = \mathbb{E} \cup \left\{ a \;\middle|\; \begin{array}{l} (isCall(e) \wedge target(e) \notin names(\mathbb{C})) \vee \\ \quad (isRet(e) \wedge caller(e) \notin names(\mathbb{C})) \wedge \\ a \in acq(e) - envActors(names(\mathbb{C})) \end{array} \right\}$, and
5. $\mathbb{U}' = \mathbb{U} \cup \{fut(e) \mid isCall(e) \wedge caller(e) \notin names(\mathbb{C})\}$.

vator class C can only be present either in its class or its boxed form. We assume that the configuration of an initial state only contains one actor-based SIOA.

Definition 9.6 (Component configuration automata):
A *component configuration automaton* $\mathcal{C}$ is a triple $\langle sioa(\mathcal{C}), config(\mathcal{C}), CL(\mathcal{C})\rangle$ where

- $sioa(\mathcal{C})$ is an SIOA;
  (As with CA, the parts of this SIOA is abbreviated to $states(\mathcal{C}) = states(sioa(\mathcal{C}))$, $start(\mathcal{C}) = start(sioa(\mathcal{C}))$, etc., for brevity.)
- a configuration mapping $config(\mathcal{C})$ with domain $states(\mathcal{C})$ such that for all $x \in states(\mathcal{C})$, $config(\mathcal{C})(x)$ is a compatible component configuration;
- $CL(\mathcal{C})$ is a set of activator classes where $\forall C \in CL(\mathcal{C}) : C \in \mathbf{CL} \cup [\mathbf{CL}]$ and $\forall C \in \mathbf{CL} : (C \in CL(\mathcal{C}) \implies [C] \notin CL(\mathcal{C})) \wedge ([C] \in CL(\mathcal{C}) \implies C \notin CL(\mathcal{C}))$;

such that the following constraints are satisfied:

1. Let $x \in start(\mathcal{C})$ and $\mathbb{C} = \langle \mathbb{A}, \mathbb{S}, \mathbb{E}, \mathbb{U}, \mathbb{B}\rangle = config(\mathcal{C})(x)$. Then, $|\mathbb{A}| = 1$, $\mathbb{U} = \emptyset$ and $(\mathcal{A}, s) \in \mathbb{C} \implies s \in start(\mathcal{A})$.
2. If $(x, e, x') \in steps(\mathcal{C})$ then $config(\mathcal{C})(x) \overset{e}{\Rightarrow}^{CL(C)} config(\mathcal{C})(x')$.
3. If $x \in states(\mathcal{C})$ and $config(\mathcal{C})(x) \overset{e}{\Rightarrow}^{CL(C)} \mathbb{C}$ for some event e and a compatible component configuration $\mathbb{C}$, then $\exists x' \in states(\mathcal{C})$ such that $config(\mathcal{C})(x') = \mathbb{C}$ and $(x, e, x') \in steps(\mathcal{C})$.
4. For all $x \in states(\mathcal{C})$
   (a) $in(\mathcal{C})(x) = in(config(\mathcal{C})(x))$
   (b) $out(\mathcal{C})(x) = out(config(\mathcal{C})(x))$
   (c) $int(\mathcal{C})(x) = int(config(\mathcal{C})(x))$

The notions of execution and traces of CCA remain the same as that of CA.

9.4 Properties of Component Configuration Automata

In each state of the CCA, each event in the signature is also exclusively classified either as an input event, an output event or an internal event as stated by the following proposition.

Proposition 9.6 (Signature disjointness of CCA):
Given a component configuration automaton $\mathcal{C}$, for all states $x \in states(\mathcal{C})$,

$$in(\mathcal{C})(x) \cap out(\mathcal{C})(x) = in(\mathcal{C})(x) \cap int(\mathcal{C})(x) = out(\mathcal{C})(x) \cap int(\mathcal{C})(x) = \emptyset \ .$$

Proof:
Follows from Definition 9.4 and Lemma 7.2. □

The fact that CCA is a generalization of ACA is captured by the following lemma. We frame this fact in terms of executions, where the executions of a CCA that has no activator classes is the same as that of an ACA, provided we strip out the component instance mapping from the component configurations. The stripping out action is represented by the function

$$stripCompConf(\langle \mathbb{A}, \mathbb{S}, \mathbb{E}, \mathbb{U}, \mathbb{B} \rangle) = \langle \mathbb{A}, \mathbb{S}, \mathbb{E}, \mathbb{U} \rangle \ ,$$

which is lifted up naturally to executions and sets of executions.

Lemma 9.3:
Let $\mathcal{C}_{ACA}$ be an ACA and $\mathcal{C}_{CCA}$ be a CCA where for each initial state in $start(\mathcal{C}_{ACA})$ there is an initial state in $start(\mathcal{C}_{CCA})$ that contains the same SIOA (identifier) mapped to the same SIOA initial state, and vice versa. Additionally, $CL(\mathcal{C}_{CCA}) = \emptyset$. Then $execs(\mathcal{C}_{ACA}) = stripCompConf(execs(\mathcal{C}_{CCA}))$.

Proof:
Follows from Definition 9.5 which is reduced to Definition 8.3 when an actor of any activator class is never created. □

The following lemma states that having activator classes does not change the intrinsic state signature of a CCA.

Lemma 9.4:
Let $\mathcal{C}_1, \mathcal{C}_2$ be CCA, $\mathbb{C}_1$ the configuration of some state x_1 of $\mathcal{C}_1$ and $\mathbb{C}_2$ the configuration of some state x_2 of $\mathcal{C}_2$ such that $names(\mathbb{C}_1) = names(\mathbb{C}_2)$ and $\forall a \in names(\mathbb{C}_1) : \exists (\mathcal{A}_1, s_1, b_1) \in \mathbb{C}_1, (\mathcal{A}_2, s_2, b_2) \in \mathbb{C}_2 : a \in names(\mathcal{A}_1) \wedge a \in names(\mathcal{A}_2) \wedge s_1(a) = s_2(a)$. Then $sig(\mathcal{C}_1)(x_1) = sig(\mathcal{C}_2)(x_2)$.

Proof:
The underlying AA of every actor $a \in names(\mathbb{C}_1)$ is the same. Hence, the signature of the AA in state $s_1(a)$ is the same as the signature of the AA in state $s_2(a)$. From Proposition 6.2, the parallel composition does not remove any event from the signature of its SIOA operands. Furthermore, no transition is lost (Definition 6.5.4).

It follows from Definitions 9.4 and 9.5, that $\widehat{sig}(\mathcal{C}_1)(x_1) = \widehat{sig}(\mathcal{C}_2)(x_2)$. From Definitions 6.5.3 and 9.4, if the signatures of the SIOA in a configuration are disjoint, an event is uniquely categorized as an input, output or internal event. Because the signature of an AA is disjoint (Lemma 7.3), it is sufficient to show that the categorization of any event $e \in \widehat{sig}(\mathcal{C}_1)(x_1)$ in $sig(\mathcal{C}_1)(x_1)$ is the same in $sig(\mathcal{C}_2)(x_2)$. Let $e \in in(\mathcal{C}_1)(x_1)$. Because one of the participating actor of e is an actor of the environment, $e \in in(\mathcal{A}_1)(s_1)$. Since $e \in \widehat{sig}(\mathcal{C}_2)(x_2)$, e can only be in $in(\mathcal{A}_2)(s_2)$ and by Definition 9.4, $e \in in(\mathcal{C}_2)(s_2)$. Similar reasoning follows for $e \in out(\mathcal{C}_1)(x_1)$. If $e \in int(\mathcal{C}_1)(x_1)$, the participating actors of e (or the only one participating actor if it involves a self call) are represented within the configuration. By Definition 9.4, $e \in int(\mathcal{C}_2)(x_2)$. By symmetry, for each event $e \in \widehat{sig}(\mathcal{C}_2)(x_2)$, $e \in \widehat{sig}(\mathcal{C}_1)(x_1)$. Because $\widehat{sig}(\mathcal{C}_1)(x_1) = \widehat{sig}(\mathcal{C}_2)(x_2)$ and the same event categorization applies, we conclude that $sig(\mathcal{C}_1)(x_1) = sig(\mathcal{C}_2)(x_2)$. □

It is important that integrating our component notion into the automaton model does not impact the observable behavior of the represented system. Regardless which classes are deemed as activator classes, each actor in the system modeled by a CCA should be able to perform interaction as modeled in an ACA. The observable behavior of an actor system is represented by a trace of the CCA. Hence, this property can be phrased as the two models having the same set of traces. To show this property, we utilize the well-known bisimulation relation ([Par81]), such that showing the existence of a bisimulation between these models implies that both ACA and CCA generate the same set of traces (cf., e.g., [BK08, Theorem 7.6]). We state below the definition of bisimulation.

Definition 9.7 (Bisimulation):
Let $\mathcal{C}_1, \mathcal{C}_2$ be configuration automata. A bisimulation for $(\mathcal{C}_1, \mathcal{C}_2)$ is a binary relation $\mathcal{R} \subseteq states(\mathcal{C}_1) \times states(\mathcal{C}_2)$ such that

1. $\forall x_1 \in start(\mathcal{C}_1) : \exists x_2 \in start(\mathcal{C}_2) : (x_1, x_2) \in \mathcal{R}$ and
   $\forall x_2 \in start(\mathcal{C}_2) : \exists x_1 \in start(\mathcal{C}_1) : (x_1, x_2) \in \mathcal{R}$;
2. $\forall (x_1, x_2) \in \mathcal{R}$:
   a) if $x_1 \overset{l}{\Rightarrow}_{\mathcal{C}_1} x_1'$, then $x_2 \overset{l}{\Rightarrow}_{\mathcal{C}_2} x_2'$ for some $x_2' \in states(\mathcal{C}_2)$ such that $(x_1', x_2') \in \mathcal{R}$ and
   b) if $x_2 \overset{l}{\Rightarrow}_{\mathcal{C}_2} x_2'$, then $x_1 \overset{l}{\Rightarrow}_{\mathcal{C}_1} x_1'$ for some $x_1' \in states(\mathcal{C}_1)$ such that $(x_1', x_2') \in \mathcal{R}$.

Intuitively, two CA $\mathcal{C}_1, \mathcal{C}_2$ are called *bisimilar* if a transition made on one CA can be matched by the other one. The definition above relates the states of the two

CA to confine when a transition of one CA can be mimicked by the other one and vice versa.

The following theorem states the desired property linking the observable behavior of ACA and CCA. Because the structure of an ACA can be fully captured by a CCA, the ACA is represented by a CCA without any activator class. The essence of the proof is that the parallel composition does not affect the observable behavior of the CCA. Because ACA do not allow CompA to be part of the configuration, we omit boxed classes in the set of activator classes of CCA.

Theorem 9.1 (Bisimulation between ACA and CCA):
Let $\mathcal{C}_1, \mathcal{C}_2$ be CCA where $CL(\mathcal{C}_1) = \emptyset$, $CL(\mathcal{C}_2) \subseteq \mathbf{CL}$, $CL(\mathcal{C}_2)$ is non-empty and for each initial state in $start(\mathcal{C}_1)$ there is an initial state in $start(\mathcal{C}_2)$ that contains the same SIOA (identifier) mapped to the same SIOA initial state, and vice versa. Then, there is a bisimulation for $(\mathcal{C}_1, \mathcal{C}_2)$.

Proof:
Without loss of generality we assume that the class of the initial actor is an activator class, and each initial state of $\mathcal{C}_1$ and $\mathcal{C}_2$ is mapped to configurations containing the same actor. If the class is not the activator class, the ACA and CCA exhibit no difference in behavior until an actor of an activator class is created. We assume the name of this actor is *init*. We drop the activator class parameter from the intrinsic transition of $\mathcal{C}_2$ to simplify the presentation because there is no need to consider them for the proof. We define a relation $\mathcal{R}$ between the states of $\mathcal{C}_1$ and $\mathcal{C}_2$ as follows.

$$\mathcal{R} = \left\{ (x_1, x_2) \;\middle|\; \begin{array}{l} x_1 \in states(\mathcal{C}_1) \wedge x_2 \in states(\mathcal{C}_2) \wedge \\ \mathbb{C}_1 = config(\mathcal{C}_1)(x_1) \wedge \mathbb{C}_2 = config(\mathcal{C}_2)(x_2) \wedge \\ exposed(\mathbb{C}_1) = exposed(\mathbb{C}_2) \wedge names(\mathbb{C}_1) = names(\mathbb{C}_2) \wedge \\ \forall a \in names(\mathbb{C}_1) : \exists \mathcal{A}_1, s_1, \mathcal{A}_2, s_2, b_2 : \\ \quad (\mathcal{A}_1, s_1, false) \in \mathbb{C}_1 \wedge a \in names(\mathcal{A}_1) \wedge \\ \quad (\mathcal{A}_2, s_2, b_2) \in \mathbb{C}_2 \wedge a \in names(\mathcal{A}_2) \wedge s_1 = s_2(a) \end{array} \right\}$$

$\mathcal{R}$ relates states that are mapped to configurations that represent the same set of actors in the same state and expose the same set of actors to the environment. Note that each state of $\mathcal{C}_1$ and $\mathcal{C}_2$ is mapped to some *compatible* configuration, ensuring the absence of degenerate cases where an intrinsic internal signature of an ACA or a CCA contains an event that only appears as an input or an output event in the SIOA part of the configuration, but not both (i.e., that event is only acknowledged by one of the two participating parties). We show that $\mathcal{R}$ is a bisimulation.

From the assumption on initial state and Definition 9.6.1, the initial configurations of $\mathcal{C}_1$ and $\mathcal{C}_2$ contain exactly the same AA $\mathcal{A}$ representing *init*. $\mathcal{R}$ then contains all pairs of states of $\mathcal{C}_1$ and $\mathcal{C}_2$ for $\mathcal{A}$ whose state mapping for $\mathcal{A}$ yields the same initial state of $\mathcal{A}$. This also ensures that $\mathcal{R}$ is not empty if the set of initial states of $\mathcal{A}$ is not empty. Therefore, the condition on initial states of the bisimulation relation is fulfilled.

Now we show the second condition of the bisimulation relation holds. Let $(x_1, x_2) \in \mathcal{R}$.

(a) Let $e \in \widehat{sig}(\mathcal{C}_1)(x_1)$ and there is x_1' such that $x_1 \overset{e}{\Rightarrow}_{\mathcal{C}_1} x_1'$. Let $a \in names(\mathbb{C}_1)$ be a participating actor of e and $s_1'(a)$ be its mapped state in $config(\mathcal{C}_1)(x_1')$. Because $(x_1, x_2) \in \mathcal{R}$, the states of a in $\mathbb{C}_1$ and in $\mathbb{C}_2$ are the same. By Lemma 9.4, $e \in \widehat{sig}(\mathcal{C}_2)(x_2)$. If a is represented in $\mathbb{C}_2$ as an AA $\mathcal{A}$, there is $s_2'(a)$ such that $s_2(a) \overset{e}{\Rightarrow}_{\mathcal{A}} s_2'(a)$. If a is part of a composed AA $\mathcal{A}_2$ in $\mathbb{C}_2$, by Definition 6.5.4 there is $s_2'(a)$ such that $s_2(a) \overset{e}{\Rightarrow}_{\mathcal{A}} s_2'(a)$. If there is more than one $s_2'(a)$, we choose the one where $s_1'(a) = s_2'(a)$. The same argument is also applied to the other participating actor a' of e, if $a' \in names(\mathbb{C}_1)$. If a' is created as the result of executing e', the resulting configuration $\mathbb{C}_2'$ is formed by adding the AA $\mathcal{A}'$ representing a' to the component configuration $\mathbb{C}_2$ or composing it with the caller's SIOA (Definition 9.5.2). The state of a' is mapped in $\mathbb{C}_2'$ to the same initial state as in $config(\mathcal{C}_1)(x_1')$ (Definition 8.3.3 and Definition 9.5.3). By Definition 9.5, there is x_2' such that $x_2 \overset{e}{\rightarrow}_{\mathcal{C}_2} x_2'$. Because the states of the other SIOA do not change, for every actor $a'' \in names(config(\mathcal{C}_1)(x_1'))$, its mapped state in $config(\mathcal{C}_1)(x_1')$ is the same as its mapped state in $config(\mathcal{C}_2)(x_2')$. Therefore, $(x_1', x_2') \in \mathcal{R}$.

(b) A similar argument applies in the other direction. The crucial point is that for a transition to happen in a composed SIOA, it has to be present in the underlying AA operands.

Thus, $\mathcal{R}$ is a bisimulation. □

Following this result, we have trace equivalence of both CCA.

Corollary 9.1:
Let $\mathcal{C}_1, \mathcal{C}_2$ be CCA where $CL(\mathcal{C}_1) = \emptyset$, $CL(\mathcal{C}_2) \subseteq$ **CL**, $CL(\mathcal{C}_2)$ is non-empty and for each initial state in $start(\mathcal{C}_1)$ there is an initial state in $start(\mathcal{C}_2)$ that contains the same SIOA (identifier) mapped to the same SIOA initial state, and vice

Corollary 9.1 (Continued)
versa. Then,

$$traces(\mathcal{C}_1) = traces(\mathcal{C}_2)$$

A useful consequence of the theorem above is that the traces are well-formed.

Corollary 9.2:
Let $\mathcal{C}$ be a CCA. Then, for each trace $t \in traces(\mathcal{C})$, t is well-formed with respect to the set of actors represented by the CCA.

It is also important that externally observable behavior of the systems are accurately represented by CCA. This behavior is captured by the external traces. Because the intrinsic signatures of ACA and CCA are handled in the same way, the classification of external events in both models is the same. As we know that the set of generated traces of both models is the same, the resulting set of external traces is also the same.

Lemma 9.5 (External behavioral equivalence of ACA and CCA):
Let $\mathcal{C}_{ACA}$ be an ACA and $\mathcal{C}_{CCA}$ be a CCA where for each initial state in $start(\mathcal{C}_{ACA})$ there is an initial state in $start(\mathcal{C}_{CCA})$ that contains the same SIOA (identifier) mapped to the same SIOA initial state, and vice versa. Then,

$$xtraces(\mathcal{C}_{ACA}) = xtraces(\mathcal{C}_{CCA}) \ .$$

Proof:
Follows from Definitions 8.2 and 9.4 and Theorem 9.1. □

9.5 Discussion

Interface of components. As mentioned in Chapter 5, the interface of a component can be over-approximated by aggregating the provided and required interface of all classes in the component. We do not apply this over-approximation to the CompA, because not all actors of a component instance are exposed to the environment and vice versa as a result of some interaction. Instead the interface through the input and output state signatures changes as more and more actors are exposed. This decision is helpful for specifying a CompA, because the specification does not have to include the description of the interface of all possible classes needed to implement the component.

Other notions of component instances. In Section 5.3, we have identified three ways to instantiate a component: static, programmer-defined and dynamic, with the dynamic kind being presented in this chapter. Adopting static component instances (actors of the same class are grouped together) requires a significant change on the overall structure of CompA and CCA. On the CompA front, the definition is more heavy duty as the internal state of each actor needs to be incorporated directly within the same CompA state. Defining configurations of CCA becomes simpler, as there is no need to present a boolean map that checks whether an SIOA represents a single actor or a group of actors. The intrinsic transitions also becomes simpler, as it is no longer necessary to distinguish whether an actor is joining a group of actors. The cost of this adoption is the loss of connection between CCA and ACA, because the behavior of single actors are inevitably incorporated in the CompA.

Adopting programmer-defined component instances (where an implementation can contain a specification in which component instance a newly created actor should be placed) has the cost of creating a partial representation of component instances that belong to the environment. Being able to state in which instance a newly created actor should be grouped means that the actor may become part of a component instance created by the environment. The definition of CompA also needs to be changed accordingly, where a creation event can now be part of the external state signatures.

CHAPTER 10

Specification of Automata

The adaptation of the DIOA model produces automata with infinite states. As stated by d'Osualdo, Kochems and Ong [DKOne], the infinite states come from these aspects:

- The dynamic topology of an actor system
- The unbounded data domain
- The unbounded dynamic creation of actors
- The unbounded capacity of the message buffers of actors

Furthermore, we consider futures and the open setting which increases the complexity. Therefore, it is necessary to obtain a finite representation of our model.

First let us note that despite the primary semantic objects of the DIOA model are the CA, as in [AL15], we do not specify the ACA or CCA directly. They are derived intrinsically from the relevant AA and CompA which are the subject of the specifications.

The proposed specification approach is inspired by the pseudocode of I/O automata described by Lynch [Lyn96, Chapter 8], a precursor to the IOA language [GLMT09]. Each specification has 3 parts describing the signatures, the states and the actions. The signatures are essentially the interfaces of the class(es) of the actor (or all possibly exposed actors of a component). The state parts focus on the internal states of the represented entity, while the actions describe the transitions taken by the entity. We assume that the data domain are described elsewhere and use the set **D** as the universe for the data domain. In addition, we assume that AA and CompA are defined such that the infinite aspects mentioned above are dealt with within their definitions. Therefore, the specifications can be finitely represented. We only need to provide a suitable interpretation of their specification.

Chapter outline. This chapter begins by giving a formal definition of the class specifications and their automaton semantics. Section 10.2 formalizes the component specification. The chapter ends with a discussion on the specification technique.

10.1 Class Specification

Class specifications, as exemplified by Figures 7.1 and 7.2, represent the expected observable behavior of class implementations. They contain the ingredients that are needed to construct the corresponding AA. The first ingredient is the set of allowed messages. In the specification, the allowed messages are given in terms of **required**, **provided**, and **internal** call and creation messages, leaving out the return messages. By instantiating the free variables in the **Allowed messages** specification according to their appropriate types, the specified **required**, **provided** and **internal** messages become the foundation of the input ($aMsg(C, in)$), output ($aMsg(C, out)$) and internal ($aMsg(C, in)$) allowed messages of the class, respectively. The set of allowed messages $aMsg(C)$ is completed by populating the set with all possible return messages based on the call and the creation messages. The definition below which describes the set of allowed messages employs the universe $\mathbf{R}(m)$ to represent the largest set of possible return values for a method call message m. This universe encompasses the data universe **D** and the actor universe **A**.

Definition 10.1 (Class allowed messages):
Given a class C, the set of allowed messages of C, $aMsg(C) \subseteq \mathbf{M}$, is the triple $\langle aMsg(C, in), aMsg(C, out), aMsg(C, int) \rangle$ where

- $\forall (a : mtd(\overline{p})) \in aMsg(C, in) : \forall v \in \mathbf{R}(a : mtd(\overline{p})) :$
  $class(a) = C \wedge (a : mtd \triangleleft v) \in aMsg(C, out)$
- $\forall (a : mtd(\overline{p})) \in aMsg(C, out) : \forall v \in \mathbf{R}(a : mtd(\overline{p})) :$
  $(a : mtd \triangleleft v) \in aMsg(C, in)$
- $\forall (a : mtd(\overline{p})) \in aMsg(C, int) : \forall v \in \mathbf{R}(a : mtd(\overline{p})) :$
  $class(a) = C \wedge (a : mtd \triangleleft v) \in aMsg(C, int)$

The second ingredient is the internal state. A class specification uses the internal state to help determine what kind of actions an actor of that class should take when it processes a message. The internal state is represented by typed *internal variables*. The universe of the types consists of the class universe **CL**, the data universe **D**, the actor universe **A**, the future universe **U** and their combination via abstract data structures. The initial value of each internal variable must be stated to ensure what kind of behavior an actor makes after it is created. For simplicity, we assume the variable names declared in the specification do not coincide with the fixed variable names of AA and the class parameters. In the following, we represent the class parameters as V_{params} and the internal variables as V_{int}. The

initial states *initStates* generated by the specification map the variables in V_{params} to all possible values and the internal variables to their specified initial values.

The third ingredient defines the expected observable behavior of an actor. As explained in Chapter 7, the observable behavior of an actor can be characterized by the events the actor generates from one release point to the next one. We name a transition from one release point to the next one an *event sequence transition*. An event sequence transition may start by generating a reaction event to some input event in the buffer, followed by a (possibly empty) sequence of output and internal emittance events. Only the last event in this sequence is allowed to be a return event following the semantics of method returns. The effect of performing these events on the state is recorded by assigning the variables to their desired values. We also include the possibility to specify blocking through the reserved variable *blkFut* of AA, where the actor waits for a specific future to be resolved. The description above is summarized by the following definition.

Definition 10.2 (Event sequence transitions):
Let V_{params} and V_{int} be sets of variables representing the parameters of the class signature and internal variables, respectively. An event sequence transition $es = \langle s, EV, s' \rangle$ where

- s, s' are states that map variables $V_{params} \cup V_{int} \cup \{blkFut\}$ to values,
- $\forall v \in V_{params} : s(v) = s'(v)$, and
- $EV = ev_1 \dots ev_n$ is a non-empty finite sequence of parameterized events such that

$$\forall i \in \{1, \dots, n\} : (i > 1 \implies isEmit(ev_i)) \wedge (i < n \implies \neg isRet(ev_i)) \, .$$

In a class specification, an event sequence transition is described by an event sequence, a precondition predicate **pre** and a state assignment **state**. The event sequence consists of parameterized events with several free variables. Variables that can be left free are variables on a reaction event and variables representing the future identities and actor names generated by the actor in an emittance event. Other variables should be part of the internal state or the class parameters. Allowing free variables to be part of a reaction event mimics the input-enabledness property of actors.

To provide some control on how an actor reacts to input events, the event sequence transition specification features a precondition predicate. The precondition predicate, specified as a first-order logic formula, states when an event sequence transition specification can be instantiated. This predicate may use the in-

ternal variables, the class parameters, and the free variables of the reaction event, if it exists in the event sequence. This predicate is evaluated on the pre-state and the input obtained from the input event the actor is reacting to. In a way, the use of this predicate restricts the capability of an actor to react to an input event beyond the actor model, because there can be some input event in the buffer of the actor model whose reaction event is not part of an event sequence transition. We will shortly see in the semantics of the class specifications how this problem is handled.

The state assignment defines what is expected of the post-state of the transition. In the specification, we use a simple assignment operator := to indicate the expected value of some variables of the post-state s' after the transition is executed. The assignments of multiple variables in one event sequence transition specification are separated by $\wedge$ and a condition can be placed on an assignment by means of an implication $\Longrightarrow$ such that the assignment happens only when the condition is fulfilled. The values of all other internal variables do not change.

The behavior of a class constructor can be specified with the keyword **constructor** before the event sequence. The event sequences of the constructor is dealt separately from the other event sequences because the constructor is only executed once. Following the restriction on constructors in αABS, where a constructor cannot contain `await` and `get` statements, the event sequence transition enacts the complete execution of a constructor. In addition, the lack of conditional checks means that the constructor can be executed in any state regardless how the internal variables and class parameters are mapped.

Instantiating the internal states, class parameters and the free variables given in the specification such that the precondition and state assignment predicates are fulfilled produces a set of event sequence transitions. As a whole, these three ingredients constitute a *class specification*. Collectively the event sequence transitions represent the *class invariant* the actor should satisfy at a release point or a blocking state. More precisely, between two consecutive release points, an actor must perform an event sequence transition specified in the class specification.

Definition 10.3 (Class specifications):
Let C be a class. A *class specification* $\mathcal{S}$ for C parameterized with an actor *this* of class C is a tuple $\langle aMsg(C), V_{params}, V_{int}, initStates, ES, ES_{cons}\rangle$ such that

- $aMsg(C)$ represents the set of allowed messages of class C,
- each $s \in initStates$ are states that map variables $V_{params} \cup V_{int}$ to some initial values and *blkFut* to $\perp$, and

Definition 10.3 (Continued)

- $ES \cup ES_{cons}$ is a set of event sequence transitions, each of these transitions satisfies the following condition:
  $\forall \langle s, ev_1 \dots ev_n, s' \rangle \in ES : \forall e \in \{ev_1 \dots ev_n\} :$
  $msg(e) \in aMsg(C) \wedge this \in \{caller(e), target(e)\}.$
  The set ES_{cons} represents the class constructor's event sequence transitions. An empty constructor is represented by $ES_{cons} = \emptyset$.

Given a class specification, we need to establish its semantics. The specification is translated to an SA that obeys the constraints of an AA, with the event sequence transitions providing the possible transitions the signature automaton may take. The translation splits each event sequence transition to AA transitions, where the internal variables immediately change their values after the first step of the event sequence transition. Such a design eases the reasoning on component behaviors. During an event sequence transition, the intermediate states should not be a release point, while the futures blocking at the release points are determined by the pre- and post-state (i.e., the s and s', respectively) of the event sequence transition. Because the event sequence transitions in the specification contain all possible mappings of the variables that satisfy the given preconditions, we only need to cherry pick those which fit to the constraints of an AA. For example, when an actor is blocked because it waits for a certain future to be resolved, only the corresponding method return emittance event can be reacted to.

The remaining question is what happens when the actor reacts to an input event at some state where no event sequence transition is applicable. Because we consider the specification to be constructive [Lam83a], the specification describes all situations that an actor can handle. Other situations not described in the specification lead to an error on the actor side. That is, reacting to that particular input event at that state produces some sort of error. To model these undesired situations, the AA are equipped with *erroneous states*. An erroneous state in AA is a state where *release* is true, but the *blkFut* is false, without any transitions that leads to a state where *release* is false. That is, the actor becomes constantly at a non-release point without any possibility for going back to a release point. Therefore, the actor cannot generate another event. Note that the actor remains input-enabled, because the execution of a transition labeled with an input event does not depend on these two variables.

Definition 10.4 formalizes these requirements. It uses the following predicate

$$sameState(s_{aut}, s_{spec}) = diffOn(s_{aut}, s_{spec}, \mathbf{V} - (V_{params} \cup V_{int} \cup \{blkFut\}))$$

to express that the state s_{aut} of the automaton is equal to the state s_{spec} of the specification.

Definition 10.4 (Class specification semantics):
Let $\langle aMsg(C), V_{params}, V_{int}, initStates, ES, ES_{cons}\rangle$ be a class specification for class C parameterized with an actor *this* of class C. Its corresponding parameterized signature automaton $\mathcal{A}(this)$ is defined by $\langle states(\mathcal{A}), start(\mathcal{A}), sig(\mathcal{A}), steps(\mathcal{A})\rangle$ where

- the domain of variables of $states(\mathcal{A})$ is the fixed variables defined in Definition 7.2 and the *params* and *ints* variables are expanded to V_{params} and V_{int}, respectively;
- $\forall s \in start(\mathcal{A})$: Constraint A2 is fulfilled and
  $\exists s' \in initStates$:
  $s(cons) = (ES_{cons} = \emptyset) \wedge$
  $s(known) = \{this\} \cup \{a \mid a \in acq(s'(v)) \wedge a \in \mathbf{A} \wedge v \in V_{params}\} \wedge$
  $\forall v \in V_{params} \cup V_{int} : s(v) = s'(v)$;
- $sig(\mathcal{A})$ is a state signature mapping such that Constraints A1 and A3 to A5 are fulfilled;
- $steps(\mathcal{A})$ is the smallest relation such that Constraints A6 to A8 are fulfilled and
  $\forall s_1 \in states(\mathcal{A}), ev_1 \in \mathbf{E}$:
  $(s_1(release) \vee (isReact(ev_1) \wedge s_1(blkFut) = fut(ev_1))) \wedge$
  $(\forall \langle s, ev_1 \ldots ev_n, s'\rangle \in ES : sameState(s_1, s) \implies$
  $\exists s'_1, s_2, s'_2, \ldots, s'_n \in states(\mathcal{A}) : sameState(s'_1, s') \wedge \forall i \in \{2, \ldots, n\} :$
  $\neg s'_{i-1}(release) \wedge diffOn(s'_{i-1}, s_i, \{buf_c, buf_r, rcvFutTgt\}) \wedge$
  $ev_i \in out(\mathcal{A})(s_i) \cup int(\mathcal{A})(s_i) \wedge s'_n(release) \implies$
  $\forall i \in \{1, \ldots n\} : (s_i, ev_i, s'_i) \in steps(\mathcal{A})) \wedge$
  $(\neg s_1(cons) \wedge \forall \langle s, ev_1 \ldots ev_n, s'\rangle \in ES_{cons} : sameState(s_1, s) \implies$
  $\exists s'_1, s_2, s'_2, \ldots, s'_n \in states(\mathcal{A}) : sameState(s'_1, s') \wedge s'_1(cons) \wedge$
  $\forall i \in \{2, \ldots, n\} :$
  $\neg s'_{i-1}(release) \wedge diffOn(s'_{i-1}, s_i, \{buf_c, buf_r, rcvFutTgt\}) \wedge$
  $ev_i \in out(\mathcal{A})(s_i) \cup int(\mathcal{A})(s_i) \wedge s'_n(release) \implies$
  $\forall i \in \{1, \ldots n\} : (s_i, ev_i, s'_i) \in steps(\mathcal{A})) \wedge$
  $(s_1(cons) \wedge \neg(\exists \langle s, ev_1 \ldots ev_n, s'\rangle \in ES : sameState(s_1, s)) \implies$
  $\exists s'_1 \in states(\mathcal{A}) : diffOn(s_1, s'_1, \{buf_c, buf_r, release, blkFut\}) \wedge$
  $\neg s'_1(release) \wedge s'_1(blkFut) = \bot \wedge (s_1, ev_1, s'_1) \in steps(\mathcal{A}))$.

The SA of a class specification has states whose class parameters and internal variables are obtained from the specification. The initial states of the SA are also obtained from the initial states of the specification by letting the actor know itself and other actors which are contained in the class parameters. The variable *cons* of an initial state is mapped to true if the constructor is empty. Otherwise, it is initialized to false, indicating that the constructor still needs to be executed. We overload the *acq* notion to work also on the mapped values of variables. The state signatures are derived from the states as regulated by the constraints of AA.

The transition relation follows the constraints of AA and is built from the event sequence transitions. Whenever the actor is in a release point, we allow an actor to execute an event sequence transition that can be executed from that release point. Similarly, if an actor is blocked for some future u, we allow the actor to execute an event sequence transition which begins with a reaction to a return event where u is resolved. The constraints of AA ensure that every transition follows the generic control flow of an actor. Because the actor is input-enabled, the intermediate state may change by virtue of receiving more input events. Thus, the definition allows some flexibility regarding the actor buffer between states s_i' and s_{i+1}.

The execution of an event sequence transition belonging to the constructors is interpreted essentially the same as for other event sequence transitions. The only difference is that we need to guarantee that the constructor is only executed once, through the use of *cons*. Following the semantics of αABS, it is possible that the constructor is not the first statement that is executed by the actor.

The non-existence of an event sequence transition that can be applied in a state indicates that the reaction to an input event causes an error. In such cases, the SA transitions to an erroneous state.

Because the constraints of AA are being used in the semantics, the corresponding SA of the specification is an AA.

Proposition 10.1:
The corresponding signature automaton of a class specification $\mathcal{S}$ of class C is an actor automaton.

Proof:
Follows from Definitions 7.2 and 10.4. □

The traces generated by the AA of a class specification follow the class invariants, because of the way the transition relations are constructed. The following lemma states this property more precisely, with the inclusion of the incomplete response to a method call. Because of the set of traces of AA is prefix-closed, we can safely focus the attention on the finite traces (cf. [LV95, Lemma 3.4]).

Lemma 10.1 (AA conformance to class invariants):
Let a be an actor of class C and $\mathcal{A}(a)$ its actor automaton obtained from a class specification $\langle aMsg(C), V_{params}, V_{int}, initStates, ES, ES_{cons}\rangle$ for class C where the parameter *this* is instantiated by a. Then, the following holds:

$$\begin{aligned}
&\forall t_a \in traces(\mathcal{A}(a)) : \exists t' : t' = t_a{\downarrow}Gen(a) \wedge |t'| \neq \infty \wedge |t'| \neq 0 \implies \\
&\quad \exists n \geq 0 : \exists \langle s^0, e_1^1 \dots e_{k_1}^1, s^1\rangle, \dots, \langle s^{n-1}, e_1^n \dots e_{k_n}^n, s^n\rangle \in ES \cup ES_{cons} : \exists t : \\
&\qquad t' = e_1^1 \dots e_{k_1}^1 \cdot \dots \cdot e_1^n \dots e_{k_n}^n \cdot t \wedge \\
&\qquad (t = [\,] \vee t = e \vee \\
&\qquad\quad (\exists t'', \langle s^n, e_1^{n+1} \dots e_{k_{n+1}}^{n+1}, s^{n+1}\rangle \in ES \cup ES_{cons} : \\
&\qquad\qquad t'' \in traces(\mathcal{A}(a)) \wedge t \in Pref(t'') \wedge t \in Pref(e_1^{n+1} \dots e_{k_{n+1}}^{n+1}) \ \wedge \\
&\qquad\qquad t''{\downarrow}Gen(a) = e_1^1 \dots e_{k_1}^1 \dots e_1^n \dots e_{k_n}^n \, e_1^{n+1} \dots e_{k_{n+1}}^{n+1}))
\end{aligned}$$

Proof (by induction on the length of t_a):
The invariant trivially holds for empty traces. Now assume that $t_a \cdot e \in traces(\mathcal{A}(a))$ where t_a satisfies the condition and its projection t' to events generated by a is finite. If t' does not end with e, then e is an input event and the condition holds by the inductive assumption. Otherwise, either t' ends with the completion of an event sequence transition, or it is still in the middle of one. If it is the former case, by Definition 10.4, e can be appended regardless whether there is an event sequence transition that matches it. If there is none, the AA moves to an erroneous state where a cannot generate any more events. In the latter case, Definition 10.4 regulates that the continuation e must be part of a matching event sequence transition. Thus, the condition is satisfied. □

The lemma above can be reformulated as follows: Every time an actor is at a release point or a blocking state, the trace generated by an actor coincides with the concatentation of the event sequences of the event sequence transitions.

Corollary 10.1:
Let a be an actor of class C and $\mathcal{A}(a)$ its actor automaton obtained from a class specification $\langle aMsg(C), V_{params}, V_{int}, initStates, ES, ES_{cons}\rangle$ for class C where the parameter *this* is instantiated by a. Then, the following holds:

$$\begin{aligned}
&\forall s \in states(\mathcal{A}(a)) : s(release) \vee s(blkFut) \neq \bot \implies \\
&\quad \exists n \geq 0 : \exists \langle s^0, e_1^1 \dots e_{k_1}^1, s^1\rangle, \dots, \langle s^{n-1}, e_1^n \dots e_{k_n}^n, s^n\rangle \in ES \cup ES_{cons} : \\
&\qquad \forall v \in V_{params} \cup V_{int} : s(v) = s^n(v) \wedge \\
&\qquad s(t_{gen}) = e_1^1 \dots e_{k_1}^1 \dots e_1^n \dots e_{k_n}^n
\end{aligned}$$

Proof:
Follows from Proposition 7.2, Def. 10.4, and Lemma 10.1.

10.2 Component Specification

Similar to a class specification, a component specification is divided into the allowed message, the internal state and the action specifications. Component specifications are typically simpler than the class specifications. Because a component specification focuses on the externally observable behavior, the description of internal messages is not needed. On the other hand, the interface of a component may consist of the interface of several classes. For example, imagine that in the server example, a client first has to set up connection with the server with some general form of queries, before it can send specific queries to the server. However, once a connection is set up, the client can use the connection to send other specific queries within the form. In this scenario, the server may perform optimization for the query computation based on the general form of the queries. The connection can be modeled, for instance, as a session actor which is created and returned by the server when the client sets up a connection. This means that there are more actors exposed to the environment of the server than just the server actor.

With this scenario in mind, a specification of the set of allowed messages for a boxed class is defined as a list of class names and their external interface. The class signature of the boxed class is also part of the specification, but the signature of other classes in the list does not need to be specified, because the creation of other actors within a component instance is internal. The information the component instance has over the environment does not change. From this specification, we can built the component's allowed messages. The parameters of the class signature of the boxed class are represented as a set of variables V_{params}. As with class specifications, the set of allowed messages contains as many messages as possible that do not breach the type restriction given in the specification.

Definition 10.5 (Component allowed messages):
Given a boxed class $[C]$, the set of allowed messages of $[C]$, $aMsg([C]) \subseteq \mathbf{M}$, is the triple $\langle aMsg([C], in), aMsg([C], out), \emptyset \rangle$ where

- $\forall (a : mtd(\overline{p})) \in aMsg([C], in) : \forall v \in \mathbf{R}(a : mtd(\overline{p})) :$
  $(a : mtd \triangleleft v) \in aMsg([C], out)$
- $\forall (a : mtd(\overline{p})) \in aMsg([C], out) : \forall v \in \mathbf{R}(a : mtd(\overline{p})) :$
  $(a : mtd \triangleleft v) \in aMsg([C], in)$

The component specifications also have a section to specify internal variables of a component instance and its initial value. As with the class specifications, the internal variables are represented as V_{int}. The initial states *initStates* of the com-

ponent contain all possible mapping of the variables of V_{params}, while all variables V_{int} are mapped to their specified initial values.

The behavior of a component instance is described by event transitions. This part differs from the class specifications, where it makes more sense there to use an event transition sequence to represent the behavior of an actor from one release point to the next one. A component instance may have more than one actor, allowing interleaving to happen. In general, the interleaving is desirable because it allows more concurrent computation to happen. As an effect, component specifications provide event transitions that highlight when a transition takes place.

An event transition specification consists of the event, the precondition when such an event may be executed, and the state assignment. In general, the specified event of a transition is either a reaction event or a method-related output emittance event. Because an output emittance event may expose actors that have not been exposed previously, such as the session actor in the extended example, the specification should allow the generation of new exposed actors. In Definition 9.1, this aspect is represented by the *expActors* variable which keeps track which actors have been exposed so far. Whenever an assignment $w :=$ **new** C appears in the **state** part of an event transition specification, an actor of class C that is not yet in *expActors* is used to in place of w. The creation assignment may only appear if the created actor is to be exposed to the environment. Because the component specification does not capture the actual creation process, the actual actor name is kept loose (i.e., non-deterministically guessed) as long as the actor's ancestors include the initial actor. The event transitions produced from the specification are the largest set of instantiations that fulfill the precondition.

Definition 10.6 (Event transitions):
Let $[C]$ be a boxed class and V_{params} and V_{int} sets of variables representing the parameters of the class signature and internal variables, respectively. An *event transition* is a triple $\langle s, ev, s' \rangle$ where

- s, s' are partial states that map variables $V_{params} \cup V_{int} \cup \{expActors\}$ to values,
- $\forall v \in V_{params} : s(v) = s'(v)$,
- $s'(expActors) - s(expActors) \subseteq acq(ev)$, and
- ev is a parameterized event.

The component specification gathers all these aspects into one unit as defined below. The initial actor of the component instance is represented by *this*.

Definition 10.7 (Component specifications):
Let $[C]$ be a boxed class. A component specification $\mathbb{S}$ for $[C]$ is a tuple $\langle aMsg([C]), V_{params}, V_{int}, initStates, ET\rangle$ parameterized with an actor *this* of class C where

- each $s \in initStates$ are partial states that map variables in $V_{params} \cup V_{int}$ to some value and *expActors* to $\emptyset$,
- $\forall \langle s, ev, s'\rangle \in ET : \exists a \in \mathbf{A} : class(a) = C \wedge$
  $(isEmit(ev) \implies msg(ev(a)) \in aMsg([C]) \wedge$
  $(isReact(ev) \implies msg(emitOf(ev(a))) \in aMsg([C], in))$.

The semantics of a component specification is a signature automaton that obeys the constraints of a CompA. The translation is similar to that for class specification, except that there are less conditions to be checked due to the event transitions and the abstraction from the internal workings of a component instance. Furthermore, potential errors need to be explicitly identified within the component specification, allowing a simpler definition of the transition relation.

Definition 10.8 (Component specification semantics):
Let $\langle aMsg([C]), V_{params}, V_{int}, initStates, ET\rangle$ be a component specification for component $[C]$. Its corresponding parameterized signature automaton $\mathcal{A}(this)$ is defined by $\langle states(\mathcal{A}), start(\mathcal{A}), sig(\mathcal{A}), steps(\mathcal{A})\rangle$ where

- The domain of variables of $states(\mathcal{A})$ is the fixed variables defined in Definition 9.1 and the *params* and *ints* variables are expanded to V_{params} and V_{int}, respectively.
- $\forall s \in start(\mathcal{A})$: Constraint C2 is fulfilled and
  $\exists s' \in initStates : s(known) = \{this\} \cup \{a \mid a = s'(v) \wedge a \in \mathbf{A} \wedge v \in V_{params}\} \wedge$
  $\forall v \in V_{params} \cup V_{int} : s(v) = s'(v)$.
- $sig(\mathcal{A})$ is a state signature mapping such that Constraints C1 and C3 to C5 are fulfilled.
- $steps(\mathcal{A})$ is the smallest relation such that Constraints C6 to C8 are fulfilled and
  $\forall s_1, s_2 \in states(\mathcal{A}), \langle s, ev, s'\rangle \in ET, a \in \mathbf{A} : class(a) = C \wedge$
  $\forall v \in V_{params} \cup V_{int} \cup \{expActors\} :$
  $s_1(v) = s(v) \wedge s_2(v) = s'(v) \implies (s_1, ev(this), s_2) \in steps(\mathcal{A})$.

The signature automaton produced by translating the component specification

is a CompA because all constraints of CompA are observed when generating the signature automaton.

Proposition 10.2:
The corresponding signature automaton of a component specification $\mathcal{S}$ of boxed class $[C]$ is a component automaton.

Proof:
Follows from Definitions 9.1 and 10.8. □

10.3 Discussion

Class vs. component specification. While the underlying specification framework is the same, the treatment for specifying AA and CompA differs. On the action front, we can define how an actor behaves from one release point to the other for AA. Specifying AA this way allows an adaptation of the class invariant verification approach of Din et al. [DJO05; DJO08; DDJO12] (Chapter 11). For CompA, generally it is not possible to specify what a component does from one release point to the next because the actors within a component act concurrently as explained in Section 9.1. Therefore, their specifications tend to be based directly on transitions that the CompA take. On the signature front, AA only represent single classes. This means that we can use the interface the class implements as the input signature part, whereas the output and internal signatures can be derived from the calls and actor creations an actor of that class may make CompA, on the other hand, generally combine the behavior of multiple classes. CompA also ignore non-reaction, internal events that are present in AA. Therefore an adjustment needs to be made to specify a CompA.

Error handling. Another difference lies on how errors are handled. For AA, input events that lead to errors (e.g., having an input parameter that causes division by zero) can be handled in a uniform way by providing a transition to a state where further release points are never reached (i.e., *release* remains true). Therefore, the conditions when errors may happen do not need to be present in the specification. For CompA, errors need to be specified explicitly, because a component instance may consist of more than one actor allowing some of the incoming input events to still be processed, even though an error happens on some parts of the component instance.

This approach of handling errors with the addition of erroneous states is also known as the *demonic* approach [NS95]. Another approach is to create a self loop

in each state representing a release point that ignores some disagreeable input, also known as the *angelic* approach. The operational semantics of αABS suggests that an actor remains input-enabled, regardless whether it encounters some errors. However, the error means that the actor cannot make any progress processing other tasks. Thus, the angelic approach is not suitable with the adaptation of DIOA model for actors.

Constructive vs. axiomatic. The specification technique used in this thesis falls into the *constructive* category [Lam83a], where the expected observable behavior of an actor or a component instance is illustrated step by step. Compared to programming languages such as αABS, we abstract from the more fine-grained operations needed to produce the events. On the component level, this abstraction is even more pronounced as we leave behind the internal events (apart from the reaction to input events). The next chapter shows how to soundly link the event-based class specifications with the class implementations.

Smith and Talcott introduce Specification Diagrams (SD) as a constructive approach to specify open actor systems [ST02]. SD provide graphical descriptions of the behavior of actors in terms of functions, where the names of the operating actors are part of the function parameters. The function body may include assertion, assumption and non-deterministic assignment statements (as in SEQ). The main difference is that a diagram may state when and what kind of a certain pattern of input messages may be received, which fits nicely with update-based languages. There are three kinds of semantics defined for Specification Diagram namely a small-step operational semantics which produces computations, a big-step operational semantics that groups together sequential computations, and an interaction path semantics derived from the computations. Only computations where no errors are raised may yield an interaction path, similar to the denotational semantics in Section 4.3 that drops all traces that yield an error. Actor creations are partially considered in these semantics, in the sense that only actors whose causal relationship with their creators is being tracked in the diagram is considered. Neither futures nor cooperative multitasking, which are present in αABS, are featured in SD. Consequently, specifying the `Worker` actors, for example, requires an extension on the underlying concepts of SD, which may significantly change the graphical constructs.

An *axiomatic* approach to specifications describes the desired properties directly [Lam83a]. For example, Din et al. [DDJO12] specify the desired trace invariants of an actor by taking projections of the trace of the actor to specific sets of futures. They also propose a more powerful dynamic logic ([HTK00]) which can describe more complex properties than trace invariants, such as when a worker

actor w in state ϕ terminates after executing a statement s, w now is in state ψ. Ahrendt and Dylla [AD12] provide more elaborate assume-guarantee style trace-based dynamic logic specifications that allow an even more specific description of how an actor performs cooperative multitasking. Logic-based specification approaches that do not feature futures are presented by Darlington and Guo [DG94] (intuitionistic linear logic [Abr93]), Dam, Fredlund and Gurov [DFG97] (first-order μ-calculus [Par76]), Duarte [Dua99] (CTL* [EH86]), Schacht [Sch01] (LTL [Pnu77]). Their approaches allow desired properties of an actor system to be compactly specified as formulas of the respective logics. Generating an automaton model that corresponds to these formulas is a challenging exercise because such automata need to be built compositionally based on the structure of the formulas, while taking the aspects of the underlying actor model into account. If the actor model is instead encoded into the formulas, the resulting formulas tend to be large and complex as illustrated by Schacht.

Session types. A different approach to statically regulate the interaction between actors is by annotating a method with a session type [THK94; DCDMY09]. Each session type describes what kind of data an actor produces and expect to receive from the environment during an interaction session. For example, the `serve` method of the `Server` class can be annotated as follows:

$$c_1 \text{ :?}\texttt{Query}.\{c_2 \text{ :!}\texttt{Query}.?\texttt{Value}\}.!\texttt{Value}$$

This means that the server is expected to receive a query, send a query, receive some value (the result of computing a query) and then send a value away. Apart from the lack of content information (for example that the query the server sends needs to be the same as the query it has received), the main issue is that session types require the use of channels. The variables c_1 and c_2 represents two different channels in which the server is communicating. The first channel is used for the clients to communicate with the server, while the second channel is used by the server to communicate with the workers. Consequently, channels become another primitive communication mechanism that needs to be present apart from the asynchronous method calls.

Part III.

Verification of Open Actor Systems

CHAPTER 11

Verification of Classes

After specifying the intended behavior of the classes and components, we need to verify their correctness. Our approach is a two-tier verification approach as shown in Figure 11.1. As stated in Chapter 1, the verification task is managed in two tiers as proposed by, for example, Misra and Chandy [MC81] and Widom et al. [WGS87]:

- Verifying that the class implementation satisfies the class specification
- Verifying that the related class and subcomponent specifications satisfy the component specification

In this chapter we focus on the verification for the first tier, where we follow the technique presented by Din et al. [DDJO12]. First a class implementation in αABS is transformed into an implementation in a simple sequential language SEQ with non-deterministic assignments, a technique proposed by Olderog and Apt [OA88]. This sequential language has well-established semantics [Apt81; Apt84] in several forms, including a weakest liberal precondition semantics ([Dij76]). The weakest liberal precondition semantics allows the verification of desired properties, which by Din et al. are formulated in terms of class invariants. The class invariant used by Din et al. generalizes the idea of pre- and post-conditions of method definitions to pre- and post-conditions between release points. In other words, they act as a *contract* between different tasks within a single actor.

In Chapter 10, we have formally described the automata-based class specifications. The class invariant is derived from a class specification by means of concatenating a chain of event sequence transitions. This derivation is reiterated in this chapter following the format used for the SEQ transformation. What is left is to link the trace semantics of the class implementation with the trace semantics of the AA of the class specification. That is, if the class invariant translated from the class specification is satisfied by the class implementation, then for any actor a of that particular class,

$$traces(a) \subseteq traces(\mathcal{A}(a)) \ .$$

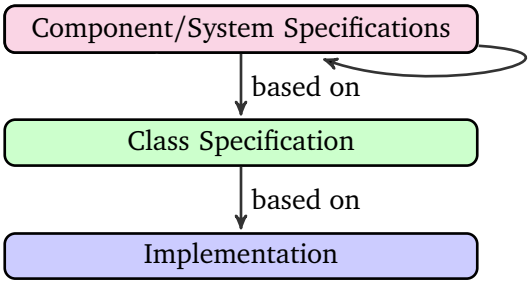


Figure 11.1.: Two-tier verification

By case analysis on the αABS statements and the well-formedness of the traces of the denotational semantics and of the AA, we establish this soundness property.
Chapter outline. This chapter is divided into 4 sections. The first section describes the SEQ language and the transformation of programs in αABS to SEQ. Section 11.2 presents the weakest liberal precondition semantics of the SEQ. The automata-based class specifications are then translated to class invariants in Section 11.3 and we show that this translation is sound with respect to the denotational semantics of αABS. Therefore, when a class implementation is shown to satisfy the class invariant, then it also satisfies the AA representation of the class specification. The verification of the second tier is discussed in the next chapter. The chapter ends with a discussion.

11.1 SEQ language

We adopt the approach of Din et al. [DJO05; DDJO12] that transforms class implementations in αABS to implementations in a simple *sequential* language SEQ enriched with a non-deterministic assignment operator. The SEQ language has a well-established semantics and a sound and relatively complete proof system [Apt81; Apt84]. In this report, the proof system is presented as weakest liberal preconditions, allowing the class to be compositionally verified method-wise. The soundness of the reasoning of the SEQ with the addition of a non-deterministic assignment operator with respect to the operational semantics of αABS hinges on the fact that there are no shared states between actors [DJO05]. Our presentation of SEQ and the encoding follows Din et al.'s [DDJO12].

The following is the syntax for SEQ statements.

$$s ::= \texttt{skip} \mid \texttt{abort} \mid \texttt{var}\ \overline{v} \mid \overline{v} := \overline{e} \mid s;s \mid \texttt{if}\ b\ \{\ s\ \}\ \texttt{else}\ \{\ s\ \} \mid \overline{y} := \texttt{some} \mid \texttt{assert}\ b \mid \texttt{assume}\ b$$

These statements represent the skip, abort, variable declarations, deterministic assignment, sequential composition, conditional, non-deterministic assignment, assert and assume statements, respectively. The abort statement is used to terminate the program, usually indicating an error. The skip, variable declaration, deterministic assignment, sequential composition and conditional statements carry the usual semantics as for αABS. The non-deterministic assignment statement assigns to the variables $\overline{v}$ some random values that match the type of the variables. The statement **assert** b indicates that some required condition b needs to be verified, whereas the statement **assume** b states some fact b. Neither statement has any effect, but they are crucial for verifying the desired behavior as we will soon see. The SEQ syntax is completed by adding procedure definitions:

$$PD ::= m(\overline{x})\ \{\ \overline{T\ y}; s\ \}$$

A procedure m accepts the parameters $\overline{x}$, allows a set of local variables $\overline{y}$ to be used when executing statement s. The declaration of local variables and the statement is called the procedure body. A procedure does not return any value (i.e., it can be seen as a **Unit** method).

SEQ also contains procedure call statements ($x := m(\overline{e})$). This is omitted from the presentation because the encoding of αABS to SEQ does not involve these statements.

To encode αABS classes in the SEQ language, the class attributes $\overline{f}$ is expanded with the variable **this** and the trace variable t, representing the self reference and the locally generated trace, respectively. The trace variable allows us to reason about the traces generated by an actor. Because of the focus on the part that is generated by the actor, it is not necessary to include the input events in the trace. They are implicitly represented via the non-deterministic assignments as described below. The fields are also extended when needed by a number of auxiliary variables ([OG76]) as needed to establish the class invariants. These auxiliary variables correspond to the internal variables used in the class specification and also extra variables introduced by the encoding. The execution of a task by an actor is represented by a SEQ process. A SEQ process operates on a state that maps the class attribute variables, the local variables and the auxiliary variables to their corresponding values. The encoding of important parts of αABS is presented in Figure 11.2.

A method definition $mtd(\overline{p})\ \{\ body\ \}$ is translated to the form $mtd(\overline{p}, \mathsf{u})\ \{\ \ll body \gg\}$ where $\ll body \gg$ is the encoding of $body$ in SEQ. The future information attached with the asynchronous method call is explicitly given in the encoding, allowing their usage when constructing the local trace. When a method call is executed by the actor, the local trace is extended by the corresponding re-

$$
\begin{array}{rcl}
\ll m(\overline{x})\{\ \overline{y}; s\ \} \gg & \stackrel{\text{def}}{=} & m(\overline{x}, fut)\{\ T\ \overline{y}, \texttt{return}; \mathsf{t} := \mathsf{t} \cdot fut \twoheadrightarrow \textbf{this} : m(\overline{x}); \\
& & \qquad \ll s \gg; \mathsf{t} := \mathsf{t} \cdot fut \leftarrow \textbf{this} : m \triangleleft \texttt{return}; \\
& & \qquad\qquad \textbf{assume}\ \textbf{wf}(\mathsf{t})\ \} \\
\ll \textbf{skip} \gg & \stackrel{\text{def}}{=} & \textbf{skip} \\
\ll s_1; s_2 \gg & \stackrel{\text{def}}{=} & \ll s_1 \gg; \ll s_2 \gg \\
\ll \textbf{if}\ e\ s_1\ \textbf{else}\ s_2 \gg & \stackrel{\text{def}}{=} & \textbf{if}\ e\ \{\ll s_1 \gg\}\ \textbf{else}\ \{\ll s_2 \gg\} \\
\ll v := e \gg & \stackrel{\text{def}}{=} & v := e \\
\ll \textbf{return}\ e \gg & \stackrel{\text{def}}{=} & \texttt{return} := e \\
\ll v := \textbf{new}\ C(\overline{e}) \gg & \stackrel{\text{def}}{=} & v' := \textbf{some}; \mathsf{t} := \mathsf{t} \cdot \textbf{this} \rightarrow v' : \textbf{new}\ C(\overline{e}); v := v'; \\
& & \qquad\qquad \textbf{assume}\ \textbf{wf}(\mathsf{t}) \\
\ll v_1 := v_2.m(\overline{e}) \gg & \stackrel{\text{def}}{=} & u' := \textbf{some}; \mathsf{t} := \mathsf{t} \cdot u' \rightarrow v_2 : m(\overline{e}); v_1 := u'; \textbf{assume}\ \textbf{wf}(\mathsf{t}) \\
\ll \textbf{await}\ u?v \gg & \stackrel{\text{def}}{=} & \textbf{assert}\ \textbf{I}(\overline{f}, \mathsf{t}) \wedge \textbf{wf}(\mathsf{t}); \overline{f}', t', v' := \textbf{some}; \\
& & t'' := u \twoheadleftarrow \mathit{getTgt}(\mathsf{t}, u) : \mathit{getMtd}(\mathsf{t}, u) \triangleleft v'; \\
& & v := v'; \mathsf{t} := \mathsf{t} \cdot t' \cdot t''; \textbf{assume}\ \textbf{I}(\overline{f}, \mathsf{t}) \wedge \textbf{wf}(\mathsf{t}) \\
\ll \textbf{await}\ e \gg & \stackrel{\text{def}}{=} & \textbf{assert}\ \textbf{I}(\overline{f}, \mathsf{t}) \wedge \textbf{wf}(\mathsf{t}); \overline{f}', t' := \textbf{some}; \\
& & \qquad\qquad \mathsf{t} := \mathsf{t} \cdot t'; \textbf{assume}\ \textbf{I}(\overline{f}, \mathsf{t}) \wedge \textbf{wf}(\mathsf{t}) \wedge e \\
\ll v = u.\textbf{get} \gg & \stackrel{\text{def}}{=} & v' := \textbf{some}; \mathsf{t} := \mathsf{t} \cdot u \twoheadleftarrow \mathit{getTgt}(\mathsf{t}, u) : \mathit{getMtd}(\mathsf{t}, u) \triangleleft v'; \\
& & \qquad\qquad v := v'; \textbf{assume}\ \textbf{wf}(\mathsf{t})
\end{array}
$$

Figure 11.2.: Encoding of αABS in SEQ

action event. When the computation is finished, the local trace is extended by the resulting method return event. A class constructor is encoded similarly to a method definition. The main difference is that it does not have the opening call reaction and closing return emittance events.

Statements that do not contribute to the cooperative multitasking aspects (i.e., **skip**, sequential composition, conditional check, variable assignment) have a straightforward translation in SEQ. Returns are modeled using the auxiliary variable `return` which is assigned to the returned value.

Creating a new actor causes the trace to be extended with the actor creation event. The identity of the new actor is guessed via the non-deterministic assignment, and the guess is assumed to be correct by assuming the well-formedness of the trace with respect to the actor *this*. The well-formedness check is represented by **wf**(t), which is an abbreviation of Definition 4.3 on well-formed traces of actors, without considering the input events. That is, **wf**(t) only confirms the well-formedness of the generated part of an actor's trace. A similar approach is

Listing 11.1: `Server` encoding in SEQ

```
// class Server()
trace t = [];
// Value serve(Query q) {
Value serve(Query q, Fut<Value> fut) {
  IWorker w;
  Fut<Value> u;
  Value v;
  // logical variables
  trace t';
  trace t'';
  IWorker w';
  Fut<Value> u';
  Value v';
  Value return;
  t = t·fut ↠ this : serve(q);
  // w = new Worker();
  w' = some;
  t = t·this → w' : new Worker();
  w = w';
  assume wf(t);
  // u = w.do(q);
  u' = some;
  t = t·u' → w : do(q);
  u = u';
  assume wf(t);
  //await u?v;
  assert I(fields, t) && wf(t);
  t' = some;
  v' = some;
  t'' = u ↞ getTgt(t, u) : getMtd(t, u) ◁ v';
  v = v';
  t = t·t'·t'';
  assume I(fields, t) && wf(t);
  //return v;
  return = v;
  t = t·fut ← this : serve ◁ return;
  assume wf(t);
}
```

also taken to deal with a method call statement, except that the guess is done on the generated future, respectively.

The **`await`** and **`get`** statements are translated in a similar way, except that **`await`** triggers a release point. In both cases, the return value is guessed and the reaction event that marks this fetching is appended to the local trace. The reaction event is built by obtaining relevant information from the trace through *getTgt* and *getMtd* functions. Because **`await`** introduces a release point, we have to check the class invariant **`I`** holds before the actor can perform other tasks. The class invariant **`I`** checks that the class attributes and the local trace satisfy the condition that must hold at release points. After which, the trace may be extended and the class attributes may change values as the actor works on other tasks. The actor then executes the rest of the method body, assuming that the class invariants hold. Ensuring that the well-formedness property holds is crucial to ensure that the result of fetching the resolved value of a future multiple times remains consistent.

We assume for the verification purpose that a SEQ process is not suspended infinitely long, implying that every method call made by this process is always resolved. This assumption allows the reasoning part to cover as many parts of the implementation as possible.

As an example, we present the encoding of the `Server` class in SEQ (Listing 11.1), where the original statements which are changed in the encoding are

$$
\begin{aligned}
wlp(\texttt{skip}, Q) &\stackrel{\text{def}}{=} Q \\
wlp(\texttt{abort}, Q) &\stackrel{\text{def}}{=} true \\
wlp(\overline{v} := \overline{e}, Q) &\stackrel{\text{def}}{=} Q^{\overline{v}}_{\overline{e}} \\
wlp(s_1; s_2, Q) &\stackrel{\text{def}}{=} wlp(s_1, wlp(s_2, Q)) \\
wlp(\texttt{if}\ b\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \}, Q) &\stackrel{\text{def}}{=} (b \wedge wlp(s_1, Q)) \vee (\neg b \wedge wlp(s_2, Q)) \\
wlp(m(\overline{x})\ body, Q) &\stackrel{\text{def}}{=} wlp(body, Q) \\
wlp(T\ y, Q) &\stackrel{\text{def}}{=} \forall y : Q \\
wlp(\overline{x} := \texttt{some}, Q) &\stackrel{\text{def}}{=} \forall \overline{x} : Q \\
wlp(\texttt{assume}\ b, Q) &\stackrel{\text{def}}{=} b \implies Q \\
wlp(\texttt{assert}\ b, Q) &\stackrel{\text{def}}{=} b \wedge Q
\end{aligned}
$$

Figure 11.3.: Weakest liberal preconditions semantics for SEQ

put as comments followed directly by their encoding. The encoding introduces to the class the generated events as they are generated. The statements that assign expressions to variables remain the same in the encoding. In the encoding, several logical variables are typically introduced to represent the values of generated by the non-deterministic assignments (e.g., t' and v'). The return statement is transformed into an assignment to the default variable `return`. These logical variables are helpful in constructing the encoding compositionally.

11.2 Weakest Liberal Preconditions

The semantics for SEQ is described by means of weakest liberal preconditions [Dij76]. This semantics allows *partial* correctness reasoning, where desired properties are established upon termination. The reasoning method offered by the weakest liberal preconditions is a predicate transformation to first-order logic. The choice of partial correctness instead of total correctness where termination also needs to be shown is motivated by the following factors:

- Actors are inherently input-enabled. Even when an actor blocks because it is waiting for a future to be resolved, it is still capable of receiving input events. From this perspective, the actor has a non-terminating behavior.
- We consider here that processing a call causes only a finite amount of output events to be produced (as implicitly indicated by the loopless syntax of αABS).

- Errors are seen as non-terminating behaviors, consistent with the first point.

First we deal with the weakest liberal preconditions of the SEQ statements which are defined in Figure 11.3. Given a statement s, the weakest liberal precondition that ensures some postcondition Q holds after executing s is denoted by $wlp(s,Q)$. The definition uses the construct $Q^{x_1 \cdot \ldots \cdot x_n}_{e_1 \cdot \ldots \cdot e_n}$, which represents the substitutions of all free occurrences of variables x_i in Q with e_i. The weakest liberal precondition for **abort** is chosen to be *true*, indicating the choice of representing errors as non-terminating computations [Mor87]. The weakest liberal preconditions for variable declarations and non-deterministic assignments are treated the same, where the universal quantifier indicates that the values of the variables are unknown in the pre-state before the statement is executed. The variable domains are implicitly extracted from the types of the variables. The weakest liberal preconditions for other statements are standard.

Example 11.2.1:
Given some class invariant $\mathbf{I}(\overline{f}, \mathsf{t})$, the serve method of the Server class has the following weakest liberal precondition.

$$
\begin{array}{l}
wlp(\texttt{serve}(q,fut),Q) = \\
\quad \forall w,u,v,w',u',t',v', \texttt{return} : w = w' \wedge u = u' \wedge v = v' = \texttt{return} \wedge \\
\qquad \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot e_2) \implies \\
\qquad \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot e_2 \cdot e_3) \implies \\
\qquad\quad (\mathbf{I}(\overline{f}, \mathsf{t} \cdot e_1 \cdot e_2 \cdot e_3) \wedge \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot e_2 \cdot e_3)) \wedge \\
\qquad\quad (\mathbf{I}(\overline{f}, \mathsf{t} \cdot e_1 \cdot e_2 \cdot e_3 \cdot t' \cdot e_4) \wedge \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot e_2 \cdot e_3 \cdot t' \cdot e_4) \implies \\
\qquad\qquad \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot e_2 \cdot e_3 \cdot t' \cdot e_4 \cdot e_5) \implies Q^{\mathsf{t}}_{\mathsf{t} \cdot e_1 \cdot e_2 \cdot e_3 \cdot t' \cdot e_4 \cdot e_5})
\end{array}
$$

where the events e_1, e_2, e_3, e_4 and e_5 are defined as follows:

- $e_1 = \langle fut \twoheadrightarrow this : \texttt{serve}(q) \rangle$
- $e_2 = \langle this \rightarrow w : \texttt{new Worker}() \rangle$
- $e_3 = \langle u \rightarrow w : \texttt{do}(q) \rangle$
- $e_4 = \langle u \twoheadleftarrow w : \texttt{do} \triangleleft v \rangle$
- $e_5 = \langle fut \leftarrow this : \texttt{serve} \triangleleft \texttt{return} \rangle$

The weakest liberal precondition above shows that the trace generated by the serve method is well-formed and the invariant is checked before and after the **await** statement is executed. During the release point that is induced by the **await** statement, we allow the local trace to be extended, indicating the progress made on the server actor's other tasks, as long as the class invariant and the well-formedness of the trace are maintained. The assignment statements appearing in the serve method are represented in the weakest liberal precondition by means of idempotence of the appropriate local and logical variables. △

The verification condition of a class C with the invariant $\mathbf{I}(\overline{f}, \mathsf{t})$ is given as follows:

$$\forall \mathsf{t}, \overline{f}, \overline{x} : \mathbf{wf}(\mathsf{t}) \wedge \mathbf{I}(\overline{f}, \mathsf{t}) \implies wlp(m(\overline{x})\ body_m, \mathbf{I}(\overline{f}, \mathsf{t}))$$

for each method definition $m(\overline{x})\ body_m$ in the class implementation. This verification condition is the same as asserting that the local trace is initially well-formed and the class invariant holds before the execution of a method call begins, after which the weakest liberal precondition of the corresponding method definition holds.

If additional knowledge with regards to certain methods needs to be proved, verification conditions with a similar form can be created. For example, if a specific pre-post condition pair $\langle P(\overline{f}, \mathsf{t}), Q(\overline{f}, \mathsf{t})\rangle$ for a method definition is needed, the verification condition that needs to be proved has the following form:

$$\forall \mathsf{t}, \overline{f}, \overline{x} : \mathbf{wf}(\mathsf{t}) \wedge P(\overline{f}, \mathsf{t}) \implies wlp(m(\overline{x})\ body_m, Q(\overline{f}, \mathsf{t}))\ .$$

In practice, the post-condition $Q(\overline{f}, \mathsf{t})$ is often enriched by the class invariant to ease the verification effort. By construction, the verification condition remains in the realm of first-order logic.

Example 11.2.2:
The verification condition of the `Server` class is

$$\begin{array}{l}
\forall \mathsf{t}, \overline{f}, \overline{x} : \mathbf{wf}(\mathsf{t}) \wedge \mathbf{I}(\overline{f}, \mathsf{t}) \implies \\
\quad \forall w, u, v, w', u', t', v', \mathtt{return} : w = w' \wedge u = u' \wedge v = v' = \mathtt{return} \ \wedge \\
\qquad \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot e_2) \implies \\
\qquad \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot e_2 \cdot e_3) \implies \\
\qquad\quad (\mathbf{I}(\overline{f}, \mathsf{t} \cdot e_1 \cdot e_2 \cdot e_3) \wedge \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot e_2 \cdot e_3)) \ \wedge \\
\qquad\quad (\mathbf{I}(\overline{f}, \mathsf{t} \cdot e_1 \cdot e_2 \cdot e_3 \cdot t' \cdot e_4) \wedge \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot e_2 \cdot e_3 \cdot t' \cdot e_4) \implies \\
\qquad\qquad \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot e_2 \cdot e_3 \cdot t' \cdot e_4 \cdot e_5) \implies \mathbf{I}(\overline{f}, \mathsf{t} \cdot e_1 \cdot e_2 \cdot e_3 \cdot t' \cdot e_4 \cdot e_5))
\end{array} \qquad \triangle$$

11.3 Class Specification to Class Invariants

Now that we have the means to verify a class implementation, we only need to reformulate the class specification in terms of class invariants. For simplicity, we assume a syntax and type checker that ensures that the allowed messages specified in a class specification matches the method signatures and creation statements of the class implementation. By matching we mean, for example, that if some method signature appears in the input or internal part of the class, the class implementation includes that method signature. We also assume that the value

of a future is retrieved only once in the implementation. Further retrievals are replaced directly with the same value.

The following definition connects the class specification to class invariants. The requirement is that the class invariant is checked just before and after a release point. The local trace should be a concatenation of event sequences defined as a chain event sequence transitions. From Corollary 10.1, we formulate this requirement in terms of the AA semantics to stay in first-order. In particular, we use state variables *release* for determining release points and t_{gen} for tracking the locally generated traces present in the AA. However, because of the same equivalence result, the verification argument can also be made directly from the class specification, particularly from the specification of the event sequence transitions.

Definition 11.1 (Class specification translation to class invariant):
Let C be a class and $\mathcal{S} = \langle aMsg(C), V_{params}, V_{int}, initStates, ES, ES_{cons}\rangle$ its class specification parameterized with an actor *this* such that $\mathcal{A}$ is the actor automaton. Given a predicate $\rho(\overline{f}, s)$ over the class parameters $\overline{f}$ and a state s of the $\mathcal{A}$, the class invariant $\mathtt{I}(\overline{f}, \mathsf{t})$ of $\mathcal{S}$ is defined as follows:

$$\mathtt{I}(\overline{f}, \mathsf{t}) \stackrel{\text{def}}{=} \exists s \in states(\mathcal{A}) : s(release) \wedge s(t_{gen}) = \mathsf{t} \wedge \rho(\overline{f}, s)$$

In the definition above, the construction of the class invariant of a class specification requires an additional predicate ρ. This user-given predicate links the class parameters used in the implementation and the variables used in the class specifications. It is typically given during the verification process as the implementation is available and only the internal variables of the specification are compared to the class parameters.

Example 11.3.1:
To verify the `Server` class, we check that verification condition on the `serve` method holds. The predicate ρ can be left as *true* because the implementation neither uses a class attribute nor has a class parameter. Let the specification of the reaction to the `serve` method call event be represented by es_1 and the specification of the reaction to the method return event by es_2. Following from weakest liberal precondition of the `serve` method, we note that the local trace generated by the execution of a `serve` method call is split into two parts, each of which conveniently matches the event sequence transitions es_1 and es_2. As the well-formedness of the traces is maintained throughout the execution, the generated futures need to be fresh. As the future generated for the method call is fresh and stored local to the method, the method return reaction event that contains the resolved value never appears in the local trace before the **`await`** statement completes its execution. Because the post-state of es_1 matches the pre-state of es_2, the verification condition holds at

every release point related to the execution of the `serve` method. Therefore, we have that the `Server` class satisfies its class specification. △

To show the soundness of the class verification, first we establish the connection between the denotational semantics and the weakest liberal precondition semantics. The following lemma states that when the class implementation is proved under the weakest liberal precondition semantics to satisfy the class invariants, all traces present in the denotation of the actor also satisfy the class invariants. The lemma is proved by analyzing the semantics case by case.

Lemma 11.1 (Denotational semantics maintains the class invariant): Let C be a class, a an actor of class C, $\mathcal{S} = \langle aMsg(C), V_{params}, V_{int}, initStates, ES, ES_{cons}\rangle$ a class specification of C, αABS (C) its implementation in αABS such that $\mathcal{S}$ is satisfied by the implementation. Then,

$$\forall t \in [\![a]\!] : \exists n : \exists \langle s^0, e_1^1 \dots e_{k_1}^1, s^1\rangle, \dots, \langle s^{n-1}, e_1^n \dots e_{k_n}^n, s^n\rangle \in ES \cup ES_{cons} : \\ removeInput(t, a) = e_1^1 \dots e_{k_1}^1 \dots e_1^n \dots e_{k_n}^n$$

where $removeInput(t, a) = t{\downarrow}Gen(a)$.

Proof:
From Corollary 10.1, we know the class invariant is equivalent to the statement above. From Lemma 4.1, $\forall t \in [\![a]\!] : t$ is well-formed with respect to $\{a\}$. Therefore, $\forall t \in [\![a]\!] :$ **wf**$(removeInput(t, a), a)$ by the definition of **wf**. The verification condition of a method definition allows any arbitrary initial trace that satisfies **wf** and the class invariant before a method definition is checked. Furthermore, the translation of the **await** statement to SEQ allows any arbitrary extension to the trace as long as the **wf** and the class invariant hold. Therefore, the trace consisting of the events generated by a method definition projected to the task should be equivalent to a trace in $[\![mtd]\!]$ (Equation (C.13)). By equivalent we mean that when the extra events *yield* and *resume* are removed from the trace of the denotation a task, these two traces are the same. Provided this equivalence holds, the lemma holds.

To check the equivalence, we check that each kind of statements produces the same change to the state. Statements that do not generate any events have the same effect on the state in the denotational semantics as defined by the weakest liberal preconditions. For the actor creation, the method call and the **get** statements, the values that are not contained in the states are guessed non-deterministically in both semantics. Because the traces are well-formed, the guesses are picked appropriately and the occurrence of method return reaction event happens

only once for each future. The effect of these statements on the state is the same as defined by the weakest liberal preconditions. Left is the **await** statement which produces a release point. The **await** statement is translated to SEQ by allowing the values of the class attributes to change non-deterministically. The values of the local variables remain the same. This change in the state is the same as what is stated in Equation (C.8). The occurrence of method return reaction event happens only once for each future. Again the effect of the **await** statement on the state is the same as defined by the weakest liberal precondition. Therefore, the changes to the state each statement in the method body produces in both weakest liberal precondition and the denotational semantics are the same. The generated traces without the extra events are also the same. Thus, the lemma holds. □

Using the result above and that the AA semantics of the class specification generates only traces that adhere to the class invariants, the class verification technique is sound with respect to the AA semantics. More precisely, when we say that a class implementation satisfies its specification, the traces generated by that implementation are traces of the AA semantics of the class specification.

Theorem 11.1 (Sound invariant translation):
Let C be a class, a an actor of class C, $\mathcal{A}(a)$ the actor automaton obtained from the class specification $\mathcal{S}$ of C, and αABS (C) its implementation in αABS. If αABS (C) satisfies the weakest liberal precondition that uses the class invariant obtained from $\mathcal{S}$, then

$$remCr([\![a]\!])\downarrow\mathbf{E} \subseteq traces(\mathcal{A}(a)) \ .$$

Proof:
From Lemmas 10.1 and 11.1, all traces in $[\![a]\!]$ and $traces(\mathcal{A}(a))$ maintain the class invariant derived from the class specification. Furthermore, these traces are also well-formed (Corollary 4.1 and Lemma 7.1). Because of the input-enabledness of $\mathcal{A}(a)$ and the well-formedness of the traces, we can choose to introduce an input event into a trace generated by $\mathcal{A}(a)$ to match the occurrence of the respective input event in the denotational semantics. From Definition 10.4, $steps(\mathcal{A}(a))$ allows an event sequence transition to occur whenever it is possible. Therefore, the $traces(\mathcal{A}(a))$ covers all traces that satisfy the class invariant derived from $\mathcal{S}$ and the theorem holds. □

11.4 Discussion

This chapter presents an adaptation of a verification technique developed by Din et al. [DDJO12] to our setting. Their technique allows local reasoning of the behavior of an actor, ideal as the basis for analyzing open actor systems. They have subsequently extended this technique to cover shared futures [DDO12a; DDO12b], providing full support for the ABS language ([JHSSS11]). Based on the weakest liberal precondition semantics for SEQ, they provide a weakest liberal precondition semantics directly for ABS. This semantics becomes the basis for proving the soundness of dynamic logic ([HTK00]) proof rules. With the same dynamic logic framework, a theorem prover for (a subset of) Java called KeY ([BHS07]) has been adapted for ABS [DOB14], enabling a semi-automatic verification of ABS programs.

In a similar line, Ahrendt and Dylla [AD12] propose a compositional, assume-guarantee proof system in dynamic logic for Creol ([JOY06]). The assume-guarantee approach allows us to say more about the context in which an actor is interacting. Despite presenting a denotational semantics for Creol, a soundness proof for the proof system is part of future work. Because the traces are based on 2-event semantics the rules are more complex.

De Boer, Clarke and Johnsen [BCJ07] present a sound and relatively complete state-based proof system for Creol enriched with shared futures. A specification for a class is defined using a combination of global and local invariants, wrapped in a Hoare logic style pre-/post-conditions. Similar to the technique of Din et al., verifying whether a class satisfies the desired invariants is performed by proving that the specification holds for each method definition. Because the specification includes global invariants, the environment must be known during the verification.

CHAPTER 12

Verification of Components

The previous chapter explains the first tier of the verification approach: the verification of class implementations. This chapter presents the second tier verification: verifying whether a component implementation satisfies a component specification. For this tier, we rely on a particular variant of the *simulation* notion [Par81] called *possibility map* [LT87; NS94]. Simulation is a relation $\mathcal{R}$ that links the states s_l of a lower-level specification with the states s_h of a higher-level specification, such that whenever $(s_l, s_h) \in \mathcal{R}$ (i.e., s_h simulates s_l), state s_h can mimic any transitions that can be taken by s_l and the resulting post-states are in $\mathcal{R}$. When such a relation exists, any trace of the lower-level specification is a trace of the higher-level specification. The possibility map relaxes this condition using the external actions in the following way. A sequence of transitions from s_l which may contain an arbitrary number of internal transitions, apart from a single external action that has to be matched, can be simulated by a single transition on s_h so that the resulting post-states are in $\mathcal{R}$. When such a map is found, the external behavior of the lower-level specification is simulated by the higher-level specification.

The notion of possibility maps fits for our class and component specifications, because the component specification concentrates on the externally observable behavior of the component instances while the implementation shows the additional internal computations to produce the externally observable behavior. The external actions are defined thanks to the clear boundary a component instance possesses. The possibility map is built on the states of two CCA, where one CCA represents the usage of the component specification (i.e., acts as the higher-level specification) while the other represents the usage of the class specification (i.e., the lower-level specification), provided the class specification has been proved. Both CCA are parameterized with the same actor as the initial actor. To facilitate compositionality, the behavior of actors that are created by the initial actor can be represented directly by verified component specifications.

Finding a possibility map is usually not a trivial task. As an aid, we identify a common fragment that a possibility map between two CCA should have. This fragment includes pairing states that contain the same set of exposed actors and the same set of input events buffered by the exposed actors.

Chapter outline. The chapter starts by discussing which (sub)component specifications are relevant for verifying a component specification using possibility map and the definition of the possibility map itself. Section 12.2 provides the guarantee that verifying the components using the possibility map is sound. This section also presents the common characteristics that need to be present in a possibility map. We end the chapter with some discussion on the approach and related work.

12.1 Possibility Maps

The base line for the verification in this second tier is that all class implementations have been verified to satisfy their respective class specifications. While it is possible to directly use the class specifications every time we want to verify some component specification (as done by Din et al. [DDJO12] on the actor level), the verification effort can be reduced if we can reuse component specifications that are already verified. Not all verified component specifications are relevant when we want to verify a component implementation. To identify the relevant ones, we collect the information of the classes of actors directly created by some actor of the component instance.

Definition 12.1 (Classes of directly created actors):
Let C be a class. The set of classes $\mathbf{C} = dirCreate(C)$ of actors that can be directly created by an instance of C is defined as

$$\{C' \mid e \in aMsg(C) \wedge isCreate(e) \wedge class(e) = C'\} .$$

Example 12.1.1:
The set of classes of actors directly created by a `Server` actor is {`Worker`}, while the set of classes of actors directly created by a `Worker` actor is {`Worker`}. △

As shown in Chapter 5, a component can be uniquely identified from its activator class. For each companion class[1] of a component, we can choose whether it will be represented by its class specification or its component specification (or both). When a set of companion classes is already represented by a subcomponent specification (i.e., they are the companion classes of a subcomponent), we can get away with using only the specification of this subcomponent. After all, a CompA never includes a creation event as part of its signature. A collection of classes and components with such categorization for a component is called a creation-complete set of subcomponents.

[1] see page 71; essentially a class that is needed by the activator class to form a component

Definition 12.2 (Creation-complete subcomponent set):
Let $\mathbf{D} = \{C_1, \ldots, C_i, [C_{i+1}], \ldots, [C_j]\} \subseteq \mathbf{CL} \cup [\mathbf{CL}]$ be a set of classes and components. We call $\mathbf{D}$ a *creation-complete set of subcomponents* with respect to an activator class $C \in \mathbf{D}$, if

$$\forall C' \in \mathbf{D} : \forall C'' \in \mathit{dirCreate}(C') \cup \mathit{dirCreate}(C) : \\ (C'' \in \mathbf{D} \implies [C''] \notin \mathbf{D}) \wedge ([C''] \in \mathbf{D} \implies C'' \notin \mathbf{D}).$$

Example 12.1.2:
Here are two creation-complete sets of subcomponents for the [`Server`] component: {`Worker`}, {[`Worker`]}. An interesting consequence of this definition is that there is only one creation-complete set of subcomponents for the [`Worker`] component: {[`Worker`]}.

The link between the specifications is the CCA. A CCA can accommodate both AA (the class specifications) and CompA (the component specifications) under the same umbrella: component configurations. Given a creation-complete set of subcomponents, we can determine whenever a CCA executes a creation event whether an actor or a component instance will be incorporated to the component configuration. Having a creation-complete set of subcomponents as the representation of a component implementation, we now have a proof obligation that a verification technique for this tier should fulfill, assuming the availability of appropriate component specifications.

Definition 12.3 (Satisfaction of component specifications):
Let

- $[C]$ be a component,
- $D = \{C_1, \ldots, C_i, [C_{i+1}], \ldots, [C_j]\}$ a creation-complete set of subcomponents with respect to C,
- $S_C, S_{C_1}, \ldots, S_{C_i}$ the respective class specifications, and
- $S_{[C]}, S_{[C_{i+1}]}, \ldots, S_{[C_j]}$ the respective component specifications.

Let $\mathcal{C}_1$ be a component configuration automaton whose set of activator classes is D and whose set of initial states are mapped to the configurations containing an actor automaton of S_C. Similarly, we define $\mathcal{C}_2$ as a component configuration automaton whose set of initial states are mapped to configurations containing a component automaton $S_{[C]}$. Then,

$$S_C, S_{C_1}, \ldots, S_{C_i}, S_{[C_{i+1}]}, \ldots, S_{[C_j]} \text{ satisfy } S_{[C]} \textit{ if } \mathit{xtraces}(\mathcal{C}_1) \subseteq \mathit{xtraces}(\mathcal{C}_2) .$$

To verify that the component specification is satisfied by the class specification and the specification of the directly created component instances, we employ the

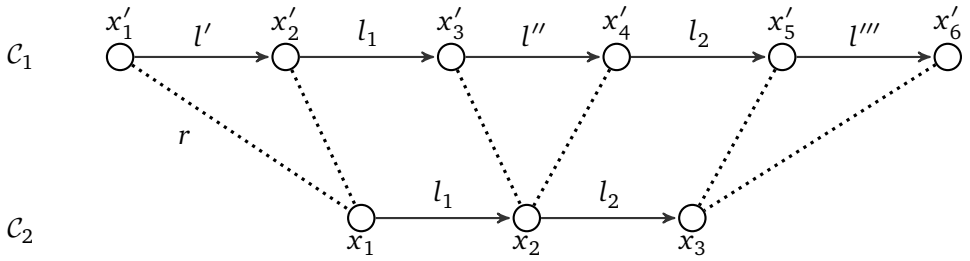


Figure 12.1.: A possibility map

possibility maps [LT87; NS94]. This notion is based on a map between states of two automata such that a step on one automaton can be simulated by a number of steps on the other automaton, similar to stuttering simulation [Man01]. The differences to stuttering simulation are the use of partial functions instead of binary relations to relate the states and that each transition on the simulated automaton cannot be classified as *internal* transitions. The latter motivates us to utilize possibility maps as our verification instrument on the component level, as the component specifications are based exclusively on the external events and their reaction events. The possibility maps reduce the global reasoning on the external traces to local reasoning on the states of the automata. The definition below formalizes the description of possibility maps.

Definition 12.4 (Possibility maps):
Let $\mathcal{C}_1, \mathcal{C}_2$ be (configuration) automata and $Act \subseteq Act(\mathcal{C}_1)$ a set of actions. A map $r = Map\langle states(\mathcal{C}_1), states(\mathcal{C}_2)\rangle$ is a *a possibility map* from $\mathcal{C}_1$ to $\mathcal{C}_2$ with respect to Act if the following conditions hold.

1. If $x \in start(\mathcal{C}_1)$ then $r(x) \neq \mathsf{undef}$ and $r(x) \in start(\mathcal{C}_2)$.
2. If $x \xrightarrow{l}_{\mathcal{C}_1} x' \wedge r(x) \neq \mathsf{undef}$ then
   - $r(x') \neq \mathsf{undef}$ and
   - either $l \notin Act \wedge r(x) = r(x')$ or $r(x) \xrightarrow{l}_{\mathcal{C}_2} r(x')$.

Figure 12.1 illustrates the notion of a possibility map on two automata $\mathcal{C}_1$ and $\mathcal{C}_2$ with respect to the set of actions $Act = \{l_1, l_2\}$. There are transitions in $\mathcal{C}_1$ that are not matched by any transition in $\mathcal{C}_2$. However, the label of these transitions is not in Act. Therefore, the state x'_3, for example, can be mapped to x_2.

12.2 Soundness of Component Verification

When a possibility map is found for the two automata, the sets of traces of the two automata form a set inclusion relation with respect to the selected set of actions. If the set of actions comprises of all actions the simulated automaton can take, this inclusion result becomes even stronger, in the sense that the simulator automaton must be able to deal with the input actions the simulated automaton can receive.

Theorem 12.1 (Soundness of possibility maps [NS94]):
Let $\mathcal{C}_1, \mathcal{C}_2$ be (configuration) automata, *Act* a set of actions and r a possibility map from $\mathcal{C}_1$ to $\mathcal{C}_2$ with respect to *Act*. Then, the following holds.

$$traces(\mathcal{C}_1)\downarrow Act \subseteq traces(\mathcal{C}_2)$$

To apply this reasoning technique to our verification problem (Definition 12.3), we need to determine an appropriate set of actions. The interest lies on the external events a component instance can take part. These external events can be generically defined by the following function given the initial actor of the component instance.

$$extEv(a) = \{e \mid isMethod(e) \wedge isEmit(e) \wedge (caller(e) \notin ancestors(a) \vee target(e) \notin ancestors(a))\}$$

In addition to the external events, the component specification utilizes the reaction events of the input events, as captured by E_{cmp} defined in Proposition 9.2.

$$E_{cmp}(a) = extEv(a) \cup \{e \mid emitOf(e) \in extEv(a)\}$$

Consequently, E_{cmp} becomes the basis for defining the possibility map. If a possibility map with respect to *envEv* between the CCA containing the component automaton of component specification and the CCA containing the actor automaton of the class specification can be found, then it shows that the component specification is satisfied by its implementation.

Lemma 12.1 (Reasoning soundness for component specification):
Let

- $[C]$ be a component,
- $D = \{C_1, \ldots, C_i, [C_{i+1}], \ldots, [C_j]\}$ a creation-complete set of subcomponents with respect to C,
- $S_C, S_{C_1}, \ldots, S_{C_i}$ the respective class specifications, and
- $S_{[C]}, S_{[C_{i+1}]}, \ldots, S_{[C_j]}$ the respective component specifications.

Lemma 12.1 (Continued)
Let $\mathcal{C}_1$ be a component configuration automaton whose set of activator classes is D and whose set of initial states are mapped to the configurations containing an actor automaton of S_C. Similarly, we define $\mathcal{C}_2$ as a component configuration automaton whose set of initial states are mapped to configurations containing a component automaton $S_{[C]}$. The initial actor of $\mathcal{C}_1$ and $\mathcal{C}_2$ is the same: a. If there exists a possibility map r from $\mathcal{C}_1$ to $\mathcal{C}_2$ with respect to $E_{cmp}(a)$,

$$xtraces(\mathcal{C}_1) \subseteq xtraces(\mathcal{C}_2) \ .$$

Proof:
From Theorem 12.1, $traces(\mathcal{C}_1)\downarrow E_{cmp}(a) \subseteq traces(\mathcal{C}_2)$. From Proposition 9.2, the set of events E overapproximates the state signatures of the CompA of $S_{[C]}$. Because the transitions of $\mathcal{C}_2$ depend exclusively on the transitions of the CompA, each trace of $\mathcal{C}_2$ consists only of events in $E_{cmp}(a)$. The external traces are obtained from the traces by projecting them to the emittance events, i.e., $extEv(a)$. Because $extEv(a) \subseteq E_{cmp}(a)$, $xtraces(\mathcal{C}_1) \subseteq xtraces(\mathcal{C}_2)$. □

Every possibility map that links two CCA shares the same properties, namely it relates only configurations that have the same set of exposed actors and input events accumulated in the buffer of the actors. Linking configurations where these properties do not hold means that different core traces are needed to reach the configurations. Because the AA and CompA also store information on the tasks the instances are working on, the paired configurations should agree on the external tasks. To ensure that the input events of both CCA process are the same, the set of received future and target actor pairs must be the same. The properties should be valid for states that are reachable from the initial states. Other (non-reachable) states may be part of the map, but because these states may never be part of the set of executions of the CCA, they do not affect the generated set of traces. In fact, it is sufficient for our verification purpose to consider maps that provide a mapping from reachable states of the first CCA to the second CCA. The following lemma sums it all.

Lemma 12.2 (Mapping prerequisites for CCA):
Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two component configuration automata such that a is the initial actor. Assuming the interface implemented by all classes is not empty, a possibility map r from $\mathcal{C}_1$ to $\mathcal{C}_2$ with respect to $E_{cmp}(a)$ fulfills the following condition:

Lemma 12.2 (Continued)

$$\textsc{PreReq} \stackrel{\text{def}}{=} \forall x \in rStates(\mathcal{C}_1) : \exists \mathbb{C}_1, \mathbb{C}_2, \mathcal{A}', s', b' : r(x) \neq \text{undef} \implies$$
$$\mathbb{C}_1 = config(\mathcal{C}_1)(x) \wedge \mathbb{C}_2 = config(\mathcal{C}_2)(x') \wedge names(\mathbb{C}_2) \subseteq names(\mathbb{C}_1) \wedge$$
$$exposed(\mathbb{C}_1) = exposed(\mathbb{C}_2) \wedge (\mathcal{A}', s', b') \in \mathbb{C}_2 \wedge$$
$$(\bigcup_{cond} s(buf_c) \cup s(buf_r)) \cap extEv(a) = s'(buf_c) \cup s(buf_r) \wedge$$
$$(\bigcup_{cond} s(tasks)) \cap \{eCore(e) \mid e \in extEv(a)\} = s'(tasks) \wedge$$
$$(\bigcup_{cond} s(rcvFutTgt)) \cap \{\langle u, a' \rangle \mid gen(u) \in ancestors(a) \wedge a' \in exposed(\mathbb{C}_1)\})$$
$$= s'(rcvFutTgt)$$

where $rStates(\mathcal{C}_1)$ is the set of (reachable) states of $\mathcal{C}_1$ to each of which there is an execution from the initial state, and

$$cond \stackrel{\text{def}}{=} (\mathcal{A}, s, b) \in \mathbb{C}_1 \wedge names(\mathcal{A}) \cap exposed(\mathbb{C}_1) \neq \emptyset \,.$$

Proof (by contradiction):
Assume that r is a map from $\mathcal{C}_1$ to $\mathcal{C}_2$ such that the PreReq is not fulfilled. Then, we need to show that r is not a possibility map (i.e., r does not satisfy Definition 12.4). This means that there is $x \in rStates(\mathcal{C}_1)$ where at least one of the conjuncts does not hold.

Case $names(\mathcal{C}_2) \supset names(\mathcal{C}_1)$: The set of actor names represented by a CompA is equivalent to the set of its exposed actors. This means that $exposed(\mathbb{C}_2) \supset exposed(\mathbb{C}_1)$. However, by Definition 9.5 there is a transition that causes an actor to be exposed in $\mathcal{C}_2$ but not in $\mathcal{C}_1$. Since x is reachable from $\mathcal{C}_1$ and the initial states all have the same set of exposed actors (i.e., the initial actor), by Definition 12.4.2 there should be x' such that $x' \xrightarrow{e}_{\mathcal{C}_1} x$ and $r(x') \xrightarrow{e}_{\mathcal{C}_2} r(x)$. However, both transitions have the same effect because $\mathcal{C}_1$ and $\mathcal{C}_2$ are both CCA. Therefore, Definition 12.4.2 cannot hold.

Case $exposed(\mathbb{C}_1) \supset exposed(\mathbb{C}_2)$: The same argument as before.

Case $(\bigcup_{cond} s(buf_c) \cup s(buf_r)) \cap extEv(a) \neq s'(buf_c) \cup s(buf_r)$: The same argument as before, with e taken as an input event.

Case $(\bigcup_{cond} s(tasks)) \cap \{eCore(e) \mid e \in extEv(a)\} \neq s'(tasks)$: The same argument as before, with e taken as a reaction to an input event.

Case $(\bigcup_{cond} s(rcvFutTgt)) \cap \{\langle u, a' \rangle \mid gen(u) \in ancestors(a) \wedge a' \in exposed(\mathbb{C}_1)\})$
$\neq s'(rcvFutTgt)$:

CCA are input enabled (Definition 9.6). Because we assume that every class is defined such that its instances can receive input messages, there can be an input event sent to a' with future u on $\mathbb{C}_1$ but not on $\mathbb{C}_2$ or vice versa. Therefore, Definition 12.4.2 does not hold.

For all cases, contradiction is reached and thus the lemma holds. □

The lemma above is applied when each class implements a non-empty interface. If class C implements an empty interface, we can always construct a map where a state in $\mathcal{C}_1$ is related to some state y in $\mathcal{C}_2$ where actors of class C have spuriously received more input events in y. Since these actors cannot receive (any further) input events that may influence their internal states (in fact, they should not receive input events to begin with), we can always pick a map where PREREQ holds. Therefore, we can use PREREQ as the basis for constructing a possibility map.

To verify whether a component specification is fulfilled, we define a map between the states of the CCA. To show the map is a possibility map, we typically use a CCA whose initial states are mapped to a configuration where the SIOA representing the initial actor is not marked as a component instance. From Theorem 9.1, this decision does not affect the observable behavior of the CCA, but it eases the comparison of the configuration without having to refer to the composed states.

To illustrate the reasoning technique, we verify the [`Worker`] component specification $\mathcal{S}_{[\texttt{Worker}]}$. The verification of the [`Server`] component specification follows the same line of reasoning.

Example 12.2.1:
Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be CCA whose initial states are mapped to a configuration containing the AA of $\mathcal{S}_{\texttt{Worker}}$ and the CompA of $\mathcal{S}_{[\texttt{Worker}]}$, respectively. The initial `Worker` actor is represented by w. We also let the component instance mapping of the AA to be set to false, allowing the creation of actors to be represented in the configuration as a separate SIOA. The map r from $\mathcal{C}_1$ to $\mathcal{C}_2$ is defined as follows:

$$
\begin{aligned}
&\text{PREREQ} \wedge \forall x \in rStates(\mathcal{C}_1), y \in states(\mathcal{C}_2) : \exists \mathcal{A}_1, \mathcal{A}_2, s_1, s_2, u, u', q : \\
&\quad (\mathcal{A}_1, s_1, false) \in config(\mathcal{C}_1)(x) \wedge (\mathcal{A}_2, s_2, true) \in config(\mathcal{C}_2)(y) \wedge \\
&\quad \{w\} = names(\mathcal{A}_1) \wedge w \in names(\mathcal{A}_2) \wedge s_2(known) = \{a \mid w \in ancestors(a)\} \wedge \\
&\quad s_2(outCalls) = s_2(genFut) = \emptyset \wedge uniqueFut(s_1(\underline{\mathsf{futTriple}})) \wedge \\
&\quad uniqueFut(s_2(\underline{\mathsf{futQPair}})) \wedge s_2(\underline{\mathsf{futQPair}}) = \{\langle u, q\rangle \mid \langle u, u', q\rangle \in s_1(\underline{\mathsf{futTriple}})\} \\
&\qquad \Longrightarrow r(x) = y
\end{aligned}
$$

where *uniqueFut* is a function that disallows the internal state of a `Worker` AA and CompA from having two elements of the same future. In other words, each

element of futTriple and futQPair can be uniquely identified by the futures (from the environment calls). The condition placed on the states of the CompA ensures that there is no ambiguity to which state of $\mathcal{C}_2$ a state of $\mathcal{C}_1$ is mapped to. This map is a possibility map as reasoned below.

0. For every $x \in states(\mathcal{C}_1)$ there is at most one $y \in states(\mathcal{C}_2)$ that fulfills the condition. This follows from the precise description of the internal state of y on the map definition.

1. $\mathcal{C}_1$ has a single initial state x_0 that is mapped to a configuration $\mathbb{C}_1$ such that $exposed(\mathbb{C}_1) = \{w\}$, $futs(\mathbb{C}_1) = \emptyset$ and $(\mathcal{A}(w), s_0, false) \in \mathbb{C}$ where $\mathcal{A}(w)$ is the AA of $\mathcal{S}_{\texttt{Worker}}$ parameterized with w and s_0 is the initial state of the worker with empty set as the value of the internal state. $\mathcal{C}_2$ also has a single initial state y_0 that is mapped to a similar configuration, except that the $\mathcal{A}(w)$ represents the CompA of $\mathcal{S}_{[\texttt{Worker}]}$. Because these two states fulfill the conditions, $r(x_0) = y_0$.

2. The related states of both CCA can receive the same input events, because of the restriction on *rcvFutTgt* of the AA/CompA representing w. Executing the reaction event $u \twoheadrightarrow w : \texttt{do}(p)$ adds the internal state in the same way such the post-states of the transition on both CCA are connected. All other events map the post-states x' to the same y. We only concentrate on the output events $e_o = u \leftarrow w : \texttt{do} \triangleleft v$. If the query q of `do` is a singleton, then v contains the expected value of $\texttt{compute}(q)$. Because x is a state reachable from some initial state, we can reason that the event sequence transitions and event transitions of the `Worker` and [`Worker`] specifications must hold in order to reach x. From $\mathcal{S}_{\texttt{Worker}}$, the transition $u \leftarrow w : \texttt{do} \triangleleft v$ if $\langle u, u', q\rangle$ is in futTriple and v is formed from $\texttt{merge}(\texttt{compute}(\texttt{firstQuery}(q), v'))$, where v' is obtained from the value stored in u'. The u' is generated alongside the call `do` on a fresh [`Worker`] instance. From $\mathcal{S}_{[\texttt{Worker}]}$, the value returned by this component instance is $\texttt{compute}(\texttt{restQuery}(q))$. From the assumption on the `merge` function, we have that $v = \texttt{compute}(q)$. Thus, the transition labeled by the output event e_o on $\mathcal{C}_1$ mimics the transition with the same label done $\mathcal{C}_2$. The Definition 12.4.2 is fulfilled.

△

Example 12.2.2:
Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be CCA whose initial states are mapped to a configuration containing the AA of $\mathcal{S}_{\texttt{Server}}$ and the CompA of $\mathcal{S}_{[\texttt{Server}]}$, respectively. The initial server actor is represented by *serv*. The map r from $\mathcal{C}_1$ to $\mathcal{C}_2$ is defined as follows:

$$\begin{aligned}&\text{PREREQ} \wedge \forall x \in rStates(\mathcal{C}_1), y \in states(\mathcal{C}_2) : \exists \mathcal{A}_1, \mathcal{A}_2, s_1, s_2, u, q : \\ &\quad (\mathcal{A}_1, s_1, false) \in config(\mathcal{C}_1)(x) \wedge (\mathcal{A}_2, s_2, true) \in config(\mathcal{C}_2)(y) \wedge \\ &\quad names(\mathcal{A}_1) = \{serv\} \wedge serv \in names(\mathcal{A}_2) \wedge s_2(known) = \{a \mid serv \in ancestors(a)\} \wedge \\ &\quad s_2(outCalls) = s_2(genFut) = \emptyset \wedge uniqueFut(s_1(\underline{\mathsf{futPair}})) \wedge \\ &\quad uniqueFut(s_2(\underline{\mathsf{futQPair}})) \wedge s_2(\underline{\mathsf{futQPair}}) = \{\langle u, q\rangle \mid \langle u, q\rangle \in s_1(\underline{\mathsf{futPair}})\} \\ &\qquad \Longrightarrow r(x) = y\end{aligned}$$

We can show that r is a possibility map by following a similar reasoning as for the [Worker] component. The main difference lies on the method calls being called on the server. Because there is no need to show how a [Worker] component instance functions, the reasoning is simpler. △

12.3 Discussion and Related Work

Reasoning on the DIOA model. To reason directly on the DIOA model, Attie and Lynch [AL15] define a trace as a sequence of external actions interleaved with external signatures. The changes in the external signatures are necessary to be present in a trace because internal actions may scramble the state signatures, making an action that is previously part of an input signature to be part of an output signature and all other possible combinations. To reason about two systems represented by CA $\mathcal{C}_1$ and $\mathcal{C}_2$, they define a specialized notion of trace merging which enables statements of the form trace t of $\mathcal{C}_1$ is a trace of $\mathcal{C}_2$. By analyzing all traces of $\mathcal{C}_1$, we obtain the result that $\mathcal{C}_1$ implements $\mathcal{C}_2$. Whether a state-based technique can be developed for the DIOA model is an open question.

Would the possibility map be applicable to the DIOA model? In the current form, the answer is negative. The problem lies in the dynamic change of the external signatures. Not only must the states match on the transition department, they also have to match the external signatures as well. In the actor setting, this problem does not appear because the signatures are stable in the sense that an event can only exclusively be part of input, output or internal signature of an AA or a CompA.

Verification using proof systems. In [KPH12], we describe a trace semantics for open actor systems and components, where futures are not present. Based on the trace semantics, we develop a Hoare-style logic that connects the input event trace of a component with the expected output event trace. The specifications are of the form $\{p\}\ D\ \{q\}$ where p and q are assertions over the input and output event traces, respectively, and D is a class C or a component $[C]$. This axiomatic specification style facilitates better representation of the desired partial behavior than the automaton specifications. However, it also carries several drawbacks:

- Relating input events that depend on certain output events requires non-trivial bookkeeping assertions, which are immediately needed when futures are introduced into the actor model.
- Providing the connection between the class specifications to the implementations is still an open problem because the verification techniques from Din et al. [DDJO12] and Ahrendt and Dylla [AD12] do not separate the traces into input and output traces.

The reasoning techniques proposed by Din et al. and Ahrendt and Dylla can be used to verify system properties. After verifying the class invariants, the system is verified by considering the composition of the class invariants for each actor that occur when the system is instantiated. The components add some structure to the system instances, allowing the verification on the system level to be done in a more compositional fashion.

De Boer [Boe02] presented a sound and relatively complete Hoare logic for concurrent processes that communicate by message passing through FIFO channels (similar to actors). He described a similar two-tier verification architecture, where the assertions are based on local and global rules. The local rules deal with the local state of a process, whereas the global rules deal with the message passing and creation of new processes. Only closed systems are considered, meaning that the environment must be explicitly present as part of the system specification. Furthermore, the global rules do not accommodate a component notion, so that all created processes have to be considered in one go.

Dam, Fredlund and Gurov [DFG97], Duarte [Dua99], Schacht [Sch01] developed a proof system for actor systems based on the respective logics (see Section 10.3). Arts and Dam, in particular, realize the proof system described by Dam, Fredlund and Gurov in form of a theorem prover for Erlang [AD99].

Verification using equivalences. As described in Section 4.4, Gaspari and Zavattaro [GZ99] and Agha and Thati [AT04] develop process algebra formalisms of the actor model. The verification method associated with the process algebra of Gaspari and Zavattaro uses asynchronous bisimulation. Agha and Thati with their Aπ-calculus goes a different way by defining a theory of *may* testing ([Hen88]). In may testing, a process is paired with some context which performs some interaction with the process. If there is at least one run where the resulting observable behavior reflects the desired behavior, then the process is said to pass the test. Agha and Thati provide such a context, where the observable behavior is given in terms of interaction paths ([Tal98], see Section 4.4). Together with Talcott, they give an Aπ-calculus characterization [TTA04] of a fragment of the specification

diagrams ([ST02]). Smith and Talcott [ST02] defined an interaction bisimulation on the specification diagrams[2] with conditions similar to the possibility maps and PREREQ. The soundness with respect to interaction path semantics is not given, but could be derived in a similar manner to our soundness result.

Verification using abstractions. A more recent development is the use of abstraction techniques to automatically model check actor systems (without the support of futures). D'Osualdo, Kochems and Ong [DKOne] consider a core Erlang program that defines a closed system and generate a Petri Net abstraction, where the actors of the same type share the same FIFO buffer. The specification is described in terms of the coverability of certain desired states. This abstraction enables them to handle unbounded creation of actors and guarantees termination of the model checking procedure.

If an actor system is known to put a bound on the depth of the interaction in terms of messages being sent around to perform a certain computation, then its representation in a π-calculus belongs to a fragment of π-calculus called depth-bounded processes [Mey08]. The running server example presented in Chapter 2 fits into this category, if the size of the query is bounded. Depth-bounded processes can be translated into well-structured transition systems ([Fin90; FS01]), for which finding if there exists a non-terminating execution is decidable. Using the translation, Zufferey et al. [ZWH12; BKWZ13] apply further abstractions on the well-structured transition systems to allow model checking closed actor systems.

Fredlund and Svensson develop a more general model checker for Erlang [FS07], where the desired properties are given in form of Büchi automata ([Büc60]) and an abstraction over the program is provided. The abstracted Erlang program is then checked under certain environmental constraints. If the resulting model has an infinite number of states, termination is not guaranteed.

An abstraction on the actor's buffer is also done by Sirjani, de Boer and Movaghar [SBM05] to allow model checking of Rebeca programs ([SJ11]) with open environment. External messages generated by the environment are assumed to be present in the buffer, so only internal messages (messages generated by actors of the systems) can be explicitly put in the buffer. When the actors of the system generate only a finite number of messages, the corresponding model is finite-state. The main constraint placed on Rebeca programs for the abstraction to work is that the network topology is static. Thus, the exposure of actors is predetermined and no actor creation is allowed.

[2] The actual term used in [ST02] is interaction simulation, a misnomer as the conditions posed on the pairs of configurations of diagrams A and B are applied in *both* directions.

CHAPTER 13

Examples

In this chapter, we provide two examples where we apply the two-tier verification approach. The examples illustrate the intricacies of open actor systems, where the environment interacts with the system beyond one-way communication where the environment only calls the head actor as exhibited by the client-server example. The first example called Ticker Factory is taken from Smith and Talcott's paper on Specification Diagrams [ST02]. This example features the exposure of actors of a component instance other than the head actor. The second example is a classical Sieve of Eratosthenes. This example features the exposure of environment actors to a component instance.

Each example starts with the description and its αABS implementation. Then, we specify the desired behavior in terms of class and component specifications. Using the class invariant technique described in Chapter 11, we verify that the class implementations satisfies the class specifications. Then we define possibility mappings for the component specifications to verify that the component implementations satisfy the component specifications.

13.1 Ticker Factory

A ticker is a monotonically increasing counter resembling a processor's clock. The ticker returns the value of the counter whenever it receives a `time` call. To produce tickers in a uniform manner, a ticker factory manages the creation process.

Listing 13.1 presents an implementation of the ticker factory in αABS. The tickers are represented by the `Ticker` class. The `Ticker` class features

- the use of a class constructor,
- an internal method that is not part of the interface it is implementing, and
- the use of class attributes.

The internal method `tick` provides the means for a ticker to independently increase the counter. The constructor initiates the first increment by calling `tick`.

The factories are represented by the `TickerFactory` class. A factory exposes a ticker to the environment each time a request for a `newTicker` is processed.

Listing 13.1: Ticker factory implementation in αABS

```
interface IFactory {
  ITicker newTicker();
}
interface ITicker { Int time(); }
class TickerFactory()
              implements IFactory {
  ITicker newTicker() {
    ITicker t = new Ticker();
    return t;
  }
}
```

```
class Ticker() implements ITicker {
  Int count = 0;
  Int time() {
    return count;
  }
  Unit tick() {
    count = count + 1;
    this.tick();
  }
  { this.tick(); }
}
```

13.1.1 Specification

Figures 13.1 to 13.3 present specifications for the `Ticker` class, the `TickerFactory` class and the [`TickerFactory`] component, respectively. The `Ticker` class specification showcases the use of the **constructor** keyword representing the class constructor's event sequence transition, while the [`TickerFactory`] component specification shows how to specify a component that exposes more than just the initial actor to the environment.

The specification $\mathcal{S}$(`Ticker`) (Figure 13.1) states that a ticker only accepts being sent `time`() calls. It also allows a ticker to internally produce a self `tick` call. The internal state in the specification consists of two variables: the counter c and lastTime which records the value of the counter the last time a `time` call comes. The lastTime variable is not used when verifying a class implementation, however it plays a crucial role in showing the monotonicity of the counter value. A ticker responded to a `time` call by returning the value of the counter and internally updating the lastTime variable. The counter evolves via `tick` internal calls. A ticker reacts to a `tick` call by increasing the internal counter and returns after making a self `tick` call. When a ticker is created, the class constructor is executed, producing a self `tick` call.

A ticker factory has a straightforward behavior: every time it is asked to supply a ticker, it creates a new one and returns the new ticker. To allow this behavior, the specification $\mathcal{S}$(`TickerFactory`) (Figure 13.2) states that a ticker factory accepts `newTicker` calls, while requiring the presence of a `Ticker` class as it creates new tickers. The internal state is empty because it does not need to store anything.

The specification $\mathcal{S}$([`TickerFactory`]) (Figure 13.3) aggregates the external behavior of a ticker factory and its tickers. A component instance of [`TickerFactory`]

Allowed messages

`Ticker`():
provided: `Int time`()
required: none
internal: `Unit tick`()

State

$\underline{c} \in$ **D**(`Int`), **initially** 0
$\underline{\mathsf{lastTime}} \in$ **D**(`Int`), **initially** 0

Actions

$u \twoheadrightarrow this : \texttt{time}() \cdot u \leftarrow this : \texttt{time} \triangleleft \underline{c}$
pre: true
state: $\underline{\mathsf{lastTime}} := \underline{c}$

constructor $u \rightarrow this : \texttt{tick}()$
pre: true
state: no change

$u \twoheadrightarrow this : \texttt{tick}() \cdot u' \rightarrow this : \texttt{tick}() \cdot u \leftarrow this : \texttt{tick} \triangleleft \texttt{Unit}$
pre: true
state: $\underline{c} := \underline{c} + 1$

Figure 13.1.: A specification of actor automaton `Ticker`(*this*) for `Ticker` class

Allowed messages

`TickerFactory`():
provided: `Ticker newTicker`()
required: `new Ticker`()
internal: none

State

none

Actions

$u \twoheadrightarrow this : \texttt{newTicker}() \cdot this \rightarrow t : \texttt{new Ticker}() \cdot u \leftarrow this : \texttt{newTicker} \triangleleft t$
pre: true
state: no change

Figure 13.2.: A specification of actor automaton `TickerFactory`(*this*) for `TickerFactory` class

Allowed messages

`TickerFactory`():		`Ticker`():	
provided:	`Ticker newTicker()`	**provided:**	`Int time()`
required:	none	**required:**	none

State

$\underline{\text{tickers}} \in \mathit{Map}\langle \mathbf{A}, \mathbf{D}(\texttt{Int})\rangle$, **initially** all actors are mapped to undef

Actions

$u \twoheadrightarrow \mathit{this} : \texttt{newTicker}()$		$u \leftarrow \mathit{this} : \texttt{newTicker} \triangleleft tk$	
pre:	true	**pre:**	$tk = \texttt{new Ticker} \wedge \underline{\text{tickers}}(tk) = \text{undef}$
state:	no change	**state:**	$\underline{\text{tickers}} := \underline{\text{tickers}}[tk \mapsto 0]$
$u \twoheadrightarrow tk : \texttt{time}()$		$u \leftarrow tk : \texttt{time} \triangleleft \underline{\text{tickers}}(tk)$	
pre:	$\underline{\text{tickers}}(tk) \neq \text{undef} \wedge \underline{\text{tickers}}(tk) \leq n$	**pre:**	true
state:	$\underline{\text{tickers}} := \underline{\text{tickers}}[tk \mapsto n]$	**state:**	no change

Figure 13.3.: A specification of component automata [`TickerFactory`](this) for [`TickerFactory`] component

exposes the created tickers. As a result, the interface of this component consists of two classes: one for `TickerFactory` and the other is for `Ticker`. The internal state of a [`TickerFactory`] instance consists of a map $\underline{\text{tickers}}$ from the created tickers to their counter value. A ticker factory accepts `newTicker` calls. The reaction to a `newTicker` call does not change the internal state of the component instance. The return event enriches the internal state by mapping the created ticker to a counter of value 0, while exposing the ticker to the environment. This value 0 corresponds to the initialized value of the counter held by a ticker. When a ticker is exposed, it can receive `time` calls. When a ticker reacts to a `time` call, the value of the counter is frozen. We guess this frozen value n, which must be at least of the previously observable value held by the ticker ($\underline{\text{tickers}}(tk)$). The ticker then uses this value to return the `time` call.

13.1.2 Class Verification

To verify the class implementation against the class specification, we encode the αABS implementation of `Ticker` and `TickerFactory` in SEQ as shown in Listing 13.2. From the encoding, we extract the weakest liberal precondition of each method in each class. These weakest liberal preconditions are shown to satisfy

Listing 13.2: `Ticker` and `TickerFactory` encoding in SEQ

```
//class Ticker()
trace t = [];
Int count = 0;
time(Fut<Int> fut) {
  Int return;
  t = t·fut ↠ this : time();
  return = count;
  t = t·fut ← this : time ◁ return;
  assume wf(t);
}
tick(Fut<Unit> fut) {
  Fut<Unit> u;
  t = t·fut ↠ this : tick();
  count = count + 1;
  u = some;
  t = t·u → this : tick();
  assume wf(t);
  t = t·fut ← this : time ◁ Unit;
  assume wf(t);
}
```

```
{
  Fut<Unit> u;
  u = some;
  t = t·u → this : tick();
  assume wf(t);
}
//class TickerFactory()
trace t = [];
newTicker(Fut<Ticker> fut) {
  Ticker tk, tk';
  Ticker return;
  t = t·fut ↠ this : newTicker();
  tk' = some;
  t = t·this → tk' : new Ticker();
  tk = tk';
  assume wf(t);
  return = tk;
  t = t·fut ← this : newTicker ◁ return;
  assume wf(t);
}
```

the class invariants extracted from the class specifications of `Ticker` (Figure 13.1) and `TickerFactory` (Figure 13.2).

The method `time` of `Ticker` has the following weakest liberal precondition:

$$wlp(\texttt{time}(fut), Q) = \forall \texttt{return} : \texttt{return} = count \wedge \texttt{wf}(\texttt{t} \cdot e_1 \cdot e_2) \implies Q^{\texttt{t}}_{\texttt{t}\cdot e_1 \cdot e_2}$$

where the events e_1 and e_2 are defined as follows:

- $e_1 = \langle fut \twoheadrightarrow this : \texttt{time}() \rangle$
- $e_2 = \langle fut \leftarrow this : \texttt{time} \triangleleft count \rangle$

The method `tick` of `Ticker` has the following weakest liberal precondition:

$$wlp(\texttt{time}(fut), Q) = \forall u : \texttt{wf}(\texttt{t} \cdot e_1 \cdot e_2) \implies \texttt{wf}(\texttt{t} \cdot e_1 \cdot e_2 \cdot e_3) \implies Q^{count \cdot \texttt{t}}_{count+1 \cdot (\texttt{t} \cdot e_1 \cdot e_2 \cdot e_3)}$$

where the events e_1, e_2 and e_3 are defined as follows:

- $e_1 = \langle fut \twoheadrightarrow this : \texttt{tick}() \rangle$
- $e_2 = \langle u \leftarrow this : \texttt{tick}() \rangle$
- $e_3 = \langle fut \leftarrow this : \texttt{tick} \triangleleft \texttt{Unit} \rangle$

The constructor of `Ticker` has the following weakest liberal precondition:

$$wlp(\texttt{Ticker}_{cons}, Q) = \forall u : \texttt{wf}(\texttt{t} \cdot u \rightarrow this : \texttt{tick}()) \implies Q^{\texttt{t}}_{\texttt{t}\cdot u \rightarrow this:\texttt{tick}()}$$

The method newTicker of TickerFactory has the following weakest liberal precondition.

$$wlp(\texttt{newTicker}(fut), Q) =$$
$$\forall tk, tk', \texttt{return} : tk = tk' = \texttt{return} \land \mathbf{wf}(\mathrm{t} \cdot e_1 \cdot e_2) \implies$$
$$\mathbf{wf}(\mathrm{t} \cdot e_1 \cdot e_2 \cdot e_3) \implies Q^{\mathrm{t}}_{\mathrm{t} \cdot e_1 \cdot e_2 \cdot e_3}$$

where the events e_1, e_2 and e_3 are defined as follows:

- $e_1 = \langle fut \twoheadrightarrow this : \texttt{newTicker}() \rangle$
- $e_2 = \langle this \rightarrow tk : \textbf{new}\ \texttt{Ticker}() \rangle$
- $e_3 = \langle fut \leftarrow this : \texttt{newTicker} \triangleleft tk \rangle$

The class invariant for the respective classes are extracted from the class specifications as described in Definition 11.1. The predicate ρ for Ticker, which connects the internal variables of the specification to the class attributes of the implementation is a simple equality $\underline{\mathtt{c}} = \mathtt{count}$, while for TickerFactory it is *true*. Plugging in the class invariant to the weakest liberal preconditions gives us the proof obligations for TickerFactory and Ticker. The remaining task is to provide the correct event sequence transitions each time the class invariant is checked. For example, the proof obligation for time is as follows:

$$\forall \mathrm{t}, \overline{f} : \mathbf{wf}(\mathrm{t}) \land \mathbf{I}(\overline{f}, \mathrm{t}) \implies \forall \texttt{return} : \texttt{return} = count \land \mathbf{wf}(\mathrm{t} \cdot e_1 \cdot e_2) \implies \mathbf{I}(\overline{f}, \mathrm{t} \cdot e_1 \cdot e_2)$$

where the local trace t is a concatenation of (instantiations of n) event sequence transitions given in Figure 13.1, such that the reaction to every tick call increases the value of the internal variable $\underline{\mathtt{c}}$ and $\underline{\mathtt{c}} = s_n(count)$. Whenever $\mathbf{I}(\overline{f}, \mathrm{t})$ takes n event sequence transitions, $\mathbf{I}(\overline{f}, \mathrm{t})$ takes $n+1$ event sequence transition with the event sequence transition involving the reaction time call. Because $\underline{\mathtt{c}}$ is the same as the value $s_n(count)$, the value returned by the ticker indeed corresponds to the value returned in the class specification.

13.1.3 System Verification

On the next tier, we show that the [TickerFactory] component implementation fulfills the its specification $\mathcal{S}_{[\texttt{TickerFactory}]}$ (Figure 13.3). To do this, we provide a possibility map between the CCA of the implementation and the component specification. The CCA of the implementation uses the class specifications $\mathcal{S}_{\texttt{Ticker}}$ and $\mathcal{S}_{\texttt{TickerFactory}}$ whose individual implementations are verified in the previous subsection. Because the classes Ticker and TickerFactory form a creation-complete set with respect to TickerFactory, the CCA contain enough ingredients to define a possibility map.

Given a head factory actor tf, let $\mathcal{C}_1$ and $\mathcal{C}_2$ be CCA whose initial states are mapped to a configuration consists of an AA specified by $\mathcal{S}_{\texttt{TickerFactory}}$ and a CompA specified by $\mathcal{S}_{[\texttt{TickerFactory}]}$, respectively. The map r from reachable states of $\mathcal{C}_1$ to states of $\mathcal{C}_2$ fulfills the following conditions:

- The prerequisite PREREQ is satisfied.
- The created actors in a reachable configuration of $\mathcal{C}_1$ can only differ by at most one actor to the set of exposed actors in the mapped configuration $\mathbb{C}_2$ of $\mathcal{C}_2$. This condition allows some flexibility when mapping a state of the ticker factory implementation where it has created a ticker and yet to expose it to its client.
- The remaining state variables of the [`TickerFactory`] component instance that is not addressed by the prerequisite is fixed appropriately.
- Whenever the ticker factory is reacting to a request to create a ticker, we allow the ticker not to be present in the tickers map of the component instance when it is not yet exposed to the environment (i.e., it is not yet returned). However, for all other mapped tickers in tickers, each must have an AA representation in the configuration of $\mathcal{C}_1$.

This map is formally defined as follows.

$$
\begin{aligned}
&\textsc{PreReq} \land \forall x \in \mathit{rStates}(\mathcal{C}_1), y \in \mathit{states}(\mathcal{C}_2) : \exists \mathcal{A}_1, \mathcal{A}_2, s_1, s_2, \mathbb{C}_1, \mathbb{C}_2 : \\
&\quad \mathbb{C}_1 = \mathit{config}(\mathcal{C}_1)(x) \land \mathbb{C}_2 = \mathit{config}(\mathcal{C}_2)(y) \land |\mathit{exposed}(\mathbb{C}_2)| + 1 \leq |\mathit{names}(\mathbb{C}_1)| \land \\
&\quad (\mathcal{A}_1, s_1, \mathit{false}) \in \mathbb{C}_1 \land (\mathcal{A}_2, s_2, \mathit{true}) \in \mathbb{C}_2 \land \mathit{names}(\mathcal{A}_1) = \{tf\} \land tf \in \mathit{names}(\mathcal{A}_2) \land \\
&\quad s_2(\mathit{known}) = \{a \mid tf \in \mathit{ancestors}(a)\} \land s_2(\mathit{outCalls}) = s_2(\mathit{genFut}) = \emptyset \land \\
&\quad (\neg s_1(\mathit{release}) \implies \\
&\qquad\qquad \exists tk \in s_1(\mathit{known}) : tk \notin \mathit{exposed}(\mathbb{C}_2) \implies s_2(\underline{\text{tickers}})(tk) = \text{undef}) \land \\
&\quad (\forall tk : s_2(\underline{\text{tickers}})(tk) \neq \text{undef} \implies \exists \mathcal{A}, s : \\
&\qquad\qquad (\mathcal{A}, s, \mathit{false}) \in \mathbb{C}_1 \land \mathit{names}(\mathcal{A}) = \{tk\} \land s(\underline{\text{lastTime}}) = s_2(\underline{\text{tickers}})(tk)) \\
&\qquad \implies r(x) = y
\end{aligned}
$$

Now we show that r is a possibility map, by showing that r is indeed a map, and the two conditions in Definition 12.4 hold.

0. For each reachable state $x \in \mathit{rStates}(\mathcal{C}_1)$ whose configuration is represented by $\mathbb{C}$, almost every ticker that is present in the configuration must be part of the internal ticker mapping tickers of the component instance and no other ticker is mapped in tickers. The only ticker that is not yet represented in tickers can appear only when the ticker factory is processing a `newTicker` call. Because of the constraints on the configurations (Definition 9.3), each ticker is represented in $\mathbb{C}$ by exactly one AA specified by $\mathcal{S}(\texttt{Ticker})$. The mapped value of these tickers corresponds to the value of the variable lastTime in the state of the AA. As $\mathbb{C}$ has exactly one represen-

tation in $states(\mathcal{C}_2)$, r is a map.

1. There exists only one initial state in both $\mathcal{C}_1$ and $\mathcal{C}_2$ representing a ticker factory that has not produced any tickers and these states are related in r.

2. For each event transition executable by $\mathcal{C}_1$ in some reachable state, we show that the mapping condition is fulfilled by analyzing each event that a ticker or a ticker factory may produce. That is, if x is a reachable state of $\mathcal{C}_1$ and $x \xrightarrow{e}_{\mathcal{C}_1} x'$ is a transition, we need to show if e is an observable generated event of a component instance of [`Sieve`], then $r(x) \xrightarrow{e}_{\mathcal{C}_2} r(x')$, otherwise $r(x) = r(x')$. For simplicity, we assume *tf* is the initial ticker factory actor and *tk* is a ticker actor that is represented in the configuration.

   Case $e = \langle u \twoheadrightarrow tf : \texttt{newTicker}() \rangle$: The internal state of the implementation representative does not change when this event is executed. Thus, the mapped state in $\mathcal{C}_2$ does not change. As the component specification defines the internal state not to change when performing a transition labeled with this event, the possibility map condition is fulfilled.

   Case $e = \langle tf \twoheadrightarrow tk : \texttt{new Ticker} \rangle$: This creation event is an internal event at the component level, so r must map the post-state of $\mathcal{C}_1$ after executing a transition labeled with this event to the same state. The ticker *tk* is at this stage only exposed to *tf* and therefore not part of the actors exposed to the environment. Because *tf* has not reached a release point, *tk* must still be mapped to undef in tickers. Thus, after executing this transition, $r(x) = r(x')$.

   Case $e = \langle u \leftarrow tf : \texttt{newTicker} \triangleleft tk \rangle$: Executing this observable generated event exposes *tk* to the environment. In $\mathcal{C}_1$, the ticker factory is back at a release point, meaning that there cannot be any discrepancy between the actors present in $\mathcal{C}_1$ and the exposed actors in $\mathcal{C}_2$. The component specification requires the post-state of the component instance to insert *tk* to tickers with the value 0. This value corresponds to the initial value of lastTime of the AA of *tk*. Because *tk* is exposed only after this event is executed, the value of lastTime still remains 0. Thus, $r(x) \xrightarrow{e}_{\mathcal{C}_2} r(x')$.

   Case $e = \langle u \twoheadrightarrow tk : \texttt{time}() \rangle$: Reacting to a `time` call changes the mapping of *tk* in tickers to some guessed value *n* larger than the previous mapped value. As *tk* can only react to such an event when it is already exposed, *tk* is part of the configuration of x. The semantics of the class specification (Definition 10.4) requires that the internal state change

of an event sequence transition to be performed immediately after the reaction event. As lastTime is assigned to the monotonically increasing counter value c, we can instantiate n to be lastTime. Because of the non-determinism introduced by the guessing, the possibility map condition holds in this case.

Case $e = \langle u \leftarrow tk : \texttt{time} \triangleleft v \rangle$: This event is an external event, so both CCA can execute a transition labeled with this event. When this transition is executed, the internal state of both the ticker actor's AA and the CompA remains the same. Thus, $r(x) \xrightarrow{e}_{\mathcal{C}_2} r(x')$.

Case $mtd(e) = \texttt{tick}$: These events are internal events that can only change the value held by the counter variable c. Because this variable plays no role in the mapping, when $\mathcal{C}_1$ makes a transition labeled by e, the post-state is mapped in r to the same state in $\mathcal{C}_2$.

Because conditions in Definition 12.4 is satisfied, r is a possibility map, and Listing 13.1 provides an implementation for $\mathcal{S}_{[\texttt{TickerFactory}]}$.

13.2 Sieve of Eratosthenes

The sieve of Eratosthenes is a well-known algorithm to find prime numbers. A typical actor-based implementation consists of a generator and a chain of filter actors. The generator has the task of sending natural numbers greater than 2 to the head of the chain. Initially, the chain consists of only one filter actor and grows longer as more numbers are fed to the chain. Each filter actor holds a prime number, with the initial filter actor holds the number 2. When a filter receives a number i and it is divisible by the number the filter holds, no further action is taken. Otherwise, it passes on the number to the next filter actor down the chain. If the filter is the last actor in the chain, a new filter actor is created, assigned i as the number it holds, and added to the end of the chain. Integral to a correct implementation is that each filter actor processes the numbers in the correct, increasing order.

Listing 13.3 presents an implementation of the sieve in αABS. In this implementation, we try to maximize concurrent processing of the integers. For simplicity, we assume that the `Int` type represents only non-negative integers. Also, we use syntactic sugar `await u?` when only the completion of a method call is of interest. This implementation consists of 2 classes `Sieve` and `Filter`, each implementing the interfaces `ISieve` and `IFilter`, respectively.

The `Sieve` class implements the generator described above. It acts as the interface to access the prime numbers, which is returned one at a time through the

Listing 13.3: An implementation of the sieve of Erastothenes in αABS

```
interface ISieve {
  Int nextPrime();
}


interface IFilter {
  Unit check(Int x);
  Unit insert(Int x);
  Int retrieve();
}

data List<T> =
  Nil |
  Cons(List<T> front, T last);

class Sieve() implements ISieve {
  IFilter head = null;
  Int count = 2;
  Fut<Int> retu = null;

  Int nextPrime() {
    Int next;
    await head != null && retu == null;
    retu = head.retrieve();
    await retu?next;
    retu = null;
    return next;
  }

  Unit generate() {
    Fut<Unit> done;
    count = count + 1;
    done = head.check(count);
    await done?;
    this.generate();
  }

  {
    head = new Filter(2, null);
    this.generate();
  }
}

class Filter(Int mine, IFilter hd)
                    implements IFilter {
  IFilter next = null;
  List<Int> todo = Nil;
  List<Int> num = Cons(Nil, mine);
  Unit check(Int x) {
    todo = Cons(todo, x);
  }
  Unit insert(Int x) {
    num = Cons(num, x);
  }
  Int retrieve() {
    Int p;
    await num != Nil;
    p = num.first();
    num = num.tail();
    return p;
  }
  Unit filter() {
    Int x;
    Fut<Unit> u;
    await todo != Nil;
    x = todo.first();
    if (x % mine != 0) {
      if (next == null) {
        if (hd == null) {
          next = new Filter(x, this);
          u = this.insert(x);
        }
        else {
          next = new Filter(x, hd);
          u = hd.insert(x);
        }
      } else {
        u = next.check(x);
      }
      await u?;
    }
    todo = todo.tail();
    this.filter();
  }
  { this.filter(); }
}
```

`nextPrime` method. The internal method `generate` implements the generation process. This method sends the integers to the head actor of the filter chain by calling the `check` method. The `generate` method progresses once the `check` method returns. At this point, the actor recursively calls `generate`. To set up the generation process correctly, the `Sieve` class is equipped with a constructor, which creates the head `Filter` actor and calls the `generate` method. Because the `generate` method is not part of the interface, this constructor and the `generate` method definition ensure that the chain obtains the integers in the right order.

The `nextPrime` method works by checking whether the head `Filter` actor has been created and the `Sieve` actor is not currently retrieving any prime numbers. This check is important because the semantics of αABS does not guarantee that the class constructor will be executed before an actor can execute incoming calls. When this guarantee is given, the condition on the `Filter` actor on Line 23 can be removed. Then it calls the head `Filter` actor to retrieve a prime number. The semantics of αABS offers no guarantee that the calls are processed in the same order as their arrival. To provide a guarantee that whenever the `Sieve` actor returns a prime number, it is the next largest prime number, the future of the `retrieve` call is stored in the field `retu`, allowing synchronization with other tasks. The return value of the `retrieve` method becomes the return value of the `nextPrime` method, and the `retu` is assigned back to **`null`**.

The `Filter` class implements the filter actors. It has two class parameters: the filter number held by the actor and the name of the head `Filter` actor. Only when the head is the actor itself does the class parameter `hd` not refer to any actor. The head `Filter` actor acts not only as the interface of the filter chain, but also as a collector of the numbers that the sieving process determines to be prime. The numbers a `Filter` actor receives are stored in two lists: `todo` and `num`. The first list is used to maintain the order of the numbers the filter chain receives from the generator, while the `num` is used to store the prime numbers, as the method definitions `check` and `insert` reflect. The `retrieve` method removes and returns the first number in `num`. As a condition to allow the retrieval, this list must not be empty.

The main filtering procedure is implemented by an internal method `filter`. In principle, it works in a similar way to the `generate` method. First it ensures that there is a number to be sifted. Then it extracts the first number `x` from the `todo` list, and checks the divisibility of this number. If `x` is divisible by `mine`, then nothing is done except removing the number from the `todo` list. If it is not divisible and the `Filter` actor is the last actor in the chain, the chain is extended with a new `Filter` actor holding `x`. Then, `x` is inserted to the `num` list of the head `Filter` actor. If the `Filter` actor is not the last actor, `x` is passed on to the next actor in

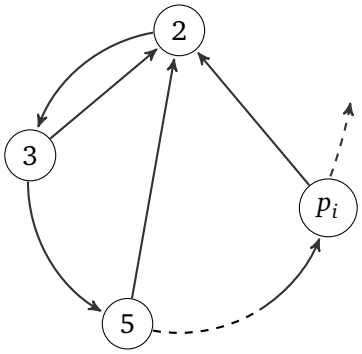


Figure 13.4.: The communication structure of the `Filter` actors

the chain. This actor then waits until the head or the next `Filter` actor returns, before continuing with the next number in the `todo` list. To initiate the filtering process, the `Filter` class is equipped with a class constructor that calls the `filter` method. The communication structure of the `Filter` chain forms somewhat like a ring, where all actors of the chain may call the head `Filter` actor, as illustrated by Section 13.2.

The implementation gives rise to two components: [`Sieve`] and [`Filter`]. Based on the communication structure of the `Filter` actors, the [`Filter`] component features the exposure of actors of the environment. From the perspective of a [`Filter`] component instance headed the filter representing the number 3, for example, the filter representing the number 2 is part of the environment. This aspect needs to be taken into consideration when specifying the desired behavior of [`Filter`] as shown in the next subsection.

13.2.1 Specification

Figures 13.5 to 13.8 present specifications for the `Sieve` class, the `Filter` class, the [`Filter`] component and the [`Sieve`] component, respectively. The [`Filter`] component features a call to the environment.

`Sieve` class specification. A specification $\mathcal{S}_{\texttt{Sieve}}$ (Figure 13.5) states that a `Sieve` actor receives only `nextPrime`() calls. As part of its interface, a `Sieve` actor may create `Filter` actors and call their `retrieve`() and `check`() methods. Internally, a `Sieve` actor may produce a self `generate` call. The internal state consists of seven variables. The variable head stores the `Filter` actor created by the `Sieve`

Allowed messages

`Sieve`():
provided: `Int nextPrime`()
required: `Int` f`.retrieve`() where $class(f) =$ `Filter`
`Unit` f`.check(Int` x`)` where $class(f) =$ `Filter`
`new Filter(Int` x`, Filter` f`)`
internal: `generate`()

State

head $\in$ **A**	**initially** `null`	tmpFut $\in$ **U**	**initially** `null`
c $\in$ **D**(`Int`)	**initially** 2	prev $\in$ **D**(`Int`)	**initially** 1
retu $\in$ **U**	**initially** `null`	tmpPrev $\in$ **D**(`Int`)	**initially** 0
nreq $\in$ **D**(`Int`)	**initially** 0		

Actions

$u \twoheadrightarrow this$: `nextPrime`()
pre: true
state: nreq $:=$ nreq $+ 1 \wedge$ tmpPrev $:= 0$

$u \rightarrow$ head : `retrieve`()
pre: head $\neq$ `null` $\wedge$ retu $=$ `null` $\wedge$ nreq > 0
state: nreq $:=$ nreq $- 1 \wedge$ retu $:= u \wedge$ tmpPrev $:= 0$

$u \twoheadleftarrow$ head : `retrieve` $\triangleleft i \cdot u' \leftarrow this$: `nextPrime` $\triangleleft i$
pre: $u =$ retu
state: retu $:=$ `null` $\wedge$ prev $:= i \wedge$ tmpPrev $:=$ prev

$u \twoheadrightarrow this$: `generate`() $\cdot u' \rightarrow$ head`.check(`c`)`
pre: head $\neq$ `null`
state: tmpPrev $:= 0 \wedge$ c $:=$ c $+ 1 \wedge$ tmpFut $:= u'$

$u \twoheadleftarrow$ head`.check` $\triangleleft$ `Unit` $\cdot u' \rightarrow this$: `generate`() $\cdot u'' \leftarrow this$: `generate` $\triangleleft$ `Unit`
pre: tmpFut $= u$
state: tmpPrev $:= 0 \wedge$ tmpFut $:=$ `null`

constructor $this \rightarrow$ head : `new Filter(2,null)` $\cdot u \rightarrow this$: `generate`()
pre: true
state: tmpPrev $:= 0$

Figure 13.5.: A specification of actor automaton `Sieve`(*this*) for `Sieve` class

actor. The variable c contains the integer to be sifted by the filter chain. nreq stores the number of `nextPrime`() calls that have been reacted to, but are still pending to retrieve a number. retu stores the future generated by the `Sieve` actor to retrieve a filtered number. The variable tmpFut temporarily stores the future of the last `check` call sent by the `Sieve` actor until the return event is reacted to. The variables prev and tmpPrev are used to store the last number returned by the `Sieve` actor, with tmpPrev acts as the intermediary value between transitions (more explanation follows). The last two variables only play some part in the [`Sieve`] component verification.

The first three event sequence transition specifications in $\mathcal{S}_{\texttt{Sieve}}$ describe how a `Sieve` actor reacts to a `nextPrime` call. First it starts by reacting to the call, which increases the number of pending requests. As done in the implementation, the `Sieve` actor must wait until the head `Filter` actor is created and no other tasks are currently retrieving any numbers. In which case, it calls head to retrieve the next filtered number and stores the generated future. The number of pending requests are decreased. Resolving the future of a `nextPrime` call is done by returning the integer i retrieved from the filter chain. The variable prev stores i. Because of the semantics of the event sequence transition specifications, this assignment happens after the `retrieve` return reaction event is executed. The intention, however, is that prev only refers to the returned value in the states *after* the return event for `nextPrime` is executed. To get around this problem, the temporary variable tmpPrev stores the previous value of the prev variable. Thus, we still have the information of the last returned number in the states between the execution of these two events.

The last three event sequence transition specifications deal with feeding of the integer sequence to the filter chain. Reacting to the `generate` method means supplying the filter chain with the next integer that needs to be filtered, represented by c. Before the `Sieve` actor can send the next integer, it has to wait until the head `Filter` actor has received the previously sent integer. By storing the future of the `check` call in tmpFut, the `Sieve` actor does not confuse itself by reacting to a different `check` return event. After which, it produces a self call. The class constructor sets up the filter chain by creating a `Filter` actor that holds the first prime number 2 and starting the feeding process.

`Filter` class specification. The specification $\mathcal{S}_{\texttt{Filter}}$ (Figure 13.6) describes the actual filtering process done by a single `Filter` actor. The `Filter` class has two parameters: myDiv representing the number the divisibility check is based on and head representing the head `Filter` actor it should send the filtered numbers to. The provided interface follows the `IFilter` interface. A `Filter` actor may call the

Allowed messages

`Filter`(`Int` myDiv, `Filter` head):

provided:	`Unit check(Int` x`)`	
	`Unit insert(Int` x`)`	
	`Int retrieve()`	
required:	f`.check(Int` x`)`	where $class(f) =$ `Filter`
	f`.insert(Int` x`)`	where $class(f) =$ `Filter`
	`new Filter(Int` x`,Filter` f`)`	
internal:	`Unit filter()`	

State

next $\in$ **A**	**initially** `null`	fStg $\in \{1,2,3\}$	**initially** 1
todo $\in$ **D**$(Seq\langle$`Int`$\rangle)$	**initially** $[\,]$	used $\subseteq$ **U**	**initially** $\emptyset$
divs $\in$ **D**$(Seq\langle$`Int`$\rangle)$	**initially** myDiv	insIdx $\in$ **D**$(Seq\langle$`Int`$\rangle)$	**initially** $[\,]$
done $\in$ **U**	**initially** `null`		

Actions

$u \twoheadrightarrow this$: `check`$(x) \cdot u \leftarrow this$: `check` $\triangleleft$ `Unit`
pre: $x > 0$
state: todo $:=$ todo $\cdot\, x$

constructor $u \rightarrow this$: `filter`$()$
pre: true
state: no change

$u \twoheadrightarrow this$: `insert`$(x) \cdot u \leftarrow this$: `insert` $\triangleleft$ `Unit`
pre: true
state: divs $:=$ divs $\cdot\, x \wedge (this \notin ancestors(gen(u)) \implies$ insIdx $:=$ insIdx $\cdot\, (|$divs$| + 1))$

$u \twoheadrightarrow this$: `retrieve`$()$
pre: true
state: no change

$u \leftarrow this$: `retrieve` $\triangleleft\, x$
pre: divs $= x \cdot$ divs$'$
state: divs $:=$ divs$' \wedge dec($insIdx$)$

$u \twoheadrightarrow this$: `filter`$()$
pre: true
state: fStg $:= 2$

$u \rightarrow this$: `filter`$() \cdot u' \leftarrow this$: `filter`$()$
pre: fStg $= 2 \wedge$ myDiv $\mid x \wedge$ todo $= x \cdot$ todo$'$
state: todo $:=$ todo$' \wedge$ fStg $:= 1$

$this \rightarrow f$: `new Filter`(todo$[1], f') \cdot u \rightarrow f'$: `insert`(x)
pre: fStg $= 2 \wedge |$todo$| > 0 \wedge$ myDiv $\nmid$ todo$[1] \wedge$ next $=$ `null` $\wedge$
$($head $=$ `null` $\implies f' = this) \wedge ($head $\neq$ `null` $\implies f' =$ head$)$
state: used $:=$ used $\cup \{u\} \wedge$ fStg $:= 3 \wedge$ done $:= u$

$u \rightarrow$ next : `check`(x)
pre: fStg $= 2 \wedge$ next $\neq$ `null` $\wedge$ todo $= x \cdot$ todo$' \wedge$ myDiv $\nmid x$
state: fStg $:= 3 \wedge$ done $:= u$

$u \twoheadleftarrow a : mtd \triangleleft$ `Unit` $\cdot\, u' \rightarrow this$: `filter`$() \cdot u'' \leftarrow this$: `filter`$()$
pre: $u =$ done $\wedge$ fStg $= 3 \wedge mtd \in \{$`insert`, `check`$\} \wedge$ todo $= x \cdot$ todo$'$
state: todo $:=$ todo$' \wedge$ fStg $:= 1 \wedge$ done $:=$ `null`

Figure 13.6.: A specification of actor automaton `Filter`$(this)$ for `Filter` class

methods `check` and `insert` on other `Filter` actors and may create new `Filter` actors. Internally, it can call the method `filter`. The internal state of the resulting actor automaton of $\mathcal{S}_{\texttt{Filter}}$ consists of seven variables. todo stores the list of incoming integers the filter needs to check. divs contains the numbers inserted to the filter in the order they are received. next represents the head of the filter chain created by the `Filter` actor. The variables done and fStg are needed to ensure a sequential connection performed by a `Filter` actor while processing a `filter` call. The next two variables used and insldx are only needed for verifying the [`Filter`] implementation. The former stores the futures of all `insert` calls made by the `Filter` actor to the head actor. The latter stores the indices of the numbers in divs inserted by some actor other than the `Filter` actor and the actors present in the chain created by the `Filter` actor. This information is required because the [`Filter`] component is verified without assuming how the environment behaves.

The event sequence transition specifications dealing with the `check` and `insert` methods are straightforward. Each appends the incoming integer to the respective lists. The `insert` method has a slight complication where it also has to store the index where the number of is inserted, if the caller is not the `Filter` actor or actors transitively created by the `Filter` actor. The specification dealing with the `retrieve` call reaction is also straightforward. Because the return event can only be generated when the actor has something to return (i.e., $|\text{divs}| > 0$), the reaction and return events are split into two event sequence transition specifications. As the `retrieve` method changes the divs list, the indices in insldx must also change. Given an integer sequence variable s, $dec(s)$ assigns s to the sequence resulting from subtracting all integers by 1 and removing the first integer if the value falls below 1 (i.e., the integer has been removed from the list). The order of the sequence is maintained.

The meat of the filtering process lies in the `filter` method. The specification states that the filtering process is done in two or three stages. The first stage is starting the filtering process by reacting to a `filter` call. The second stage is on only when the todo list is not empty and it is at a stage where the filtering process can happen ($\text{fStg} = 2$). When the first number x in the todo list is divisible by myDiv (represented by $\text{myDiv} \mid x$[1]), nothing further is done except removing x from the list and returning to the first filter stage. Otherwise ($\text{myDiv} \nmid x$), there are two cases to handle. If the `Filter` actor is not the last actor in the chain ($\text{next} \neq \texttt{null}$), it passes on the number to the next `Filter` actor on the chain by calling the `check` method. Otherwise, it creates a new `Filter` actor assigned with the number x before sending x to the head actor of the chain. A distinction needs

[1] read as myDiv divides x

to be made whether the Filter actor is the head actor or some other actor given in head. The future of a check or a insert call is temporarily stored in done and the actor progresses to the third filter stage. When the check or insert calls return (confirmed by comparing the future of the reaction event with done), a recursive filter call is made to continue the filtering process, x is removed from the list, the temporary storage for the future is nullified, and the Filter actor is back to the first filter stage. Without temporarily storing the future of check or insert calls, the Filter actor can actually progress to filter the next number, without properly ensuring that the number is passed on to the next Filter actor on the chain or inserted in the right order.

[Filter] **component specification.** The expected behavior of a filter chain is represented by a specification $\mathcal{S}_{[\texttt{Filter}]}$ (Figure 13.7). Essentially, we want the chain to sift the given integers, based on the initial divisor provided by the class parameter myDiv. However, the specification needs to consider the behavior of the component instance when it represents the *tail* of some chain (i.e., the class parameter head is filled with some Filter actor). Furthermore, because of the environment assumption (more precisely, lack thereof), we also need to cover the situation where integers are insert-ed to the filtered list, even though the component instance does not represent the whole chain.

The provided interface of [Filter] follows the IFilter interface. The required interface only consists of the insert call, to reflect that the component instance represents only the chain tail. The internal state of a component automaton of $\mathcal{S}_{[\texttt{Filter}]}$ is represented by 4 variables. The variable kept contains the filtered integers, based on the integers the component instance needs to check. The variable sifted represents these integers in a sequence form. However, as the filtered integers may be retrieved or inserted to the head filter actor, this list may consist of the remaining filtered integers. The variable divs stores all integers inserted to the component instance via the insert call. The variable used is the counterpart of the used variable in $\mathcal{S}_{\texttt{Filter}}$.

The event transition specifications that deal with the reaction to a check, insert or retrieve call are straightforward. The return of a retrieve call is a bit more tricky to specify. In general, the return value should come from the list of inserted integers divs. However, when a component instance represents the complete chain, as reflected by the condition head $=$ `null`, the return value can also come from the sifted list. In this specification, the resulting component automaton may make a non-deterministic decision from which list the return value is extracted. When the component instance does not represent a complete chain, it must pass on the filtered integers to the head actor. This is done through a

Allowed messages

`Filter`(**`Int`** myDiv, `Filter` head):
provided: **`Unit`** `check`(**`Int`** x)
`Unit` `insert`(**`Int`** x)
`Int` `retrieve`()
required: **`Unit`** f.`insert`(**`Int`** x) where $class(f) =$ `Filter`

State

kept $\subseteq$ **D**(**`Int`**)	**initially** {myDiv}
sifted $\in$ **D**($Seq\langle$**`Int`**$\rangle$)	**initially** {myDiv}
divs $\in$ **D**($Seq\langle$**`Int`**$\rangle$)	**initially** $\emptyset$
used $\subseteq$ **U**	**initially** $\emptyset$

Actions

$u \twoheadrightarrow this$: `check`(**`Int`** x)
pre: $x > 0 \wedge \forall y \in$ kept $: y \nmid x$
state: sifted $:=$ sifted $\cdot\, x \wedge$ kept $:=$ kept $\cup \{x\}$

$u \twoheadrightarrow this$: `check`(**`Int`** x)
pre: $\exists y \in$ kept $: y \mid x$
state: no change

$u \leftarrow this$: `check` $\triangleleft$ **`Unit`**
pre: true
state: no change

$u \twoheadrightarrow this$: `insert`(**`Int`** x)
pre: true
state: divs $:=$ divs $\cdot\, x$

$u \leftarrow this$: `insert` $\triangleleft$ **`Unit`**
pre: true
state: no change

$u \twoheadrightarrow this$: `retrieve`(**`Int`** x)
pre: true
state: no change

$u \leftarrow this$: `retrieve` $\triangleleft\, x$
pre: divs $= x \cdot$ divs$'$
state: divs $:=$ divs$'$

$u \leftarrow this$: `retrieve` $\triangleleft\, x$
pre: head $=$ **`null`** $\wedge$ sifted $= x \cdot$ sifted$'$
state: sifted $:=$ sifted$'$

$u \rightarrow$ head : `insert`(sifted[1])
pre: head $\neq$ **`null`** $\wedge$ |sifted| > 0
state: used $:=$ used $\cup \{u\}$

$u \twoheadleftarrow$ head : `insert` $\triangleleft$ **`Unit`**
pre: $u \in$ used $\wedge$ sifted $= x \cdot$ sifted$'$
state: sifted $:=$ sifted$'$ $\wedge$
used $:=$ used $- \{u\}$

Figure 13.7.: A specification of component automaton [`Filter`](*this*) for [`Filter`] component

Allowed messages

`Sieve`():
provided: `nextPrime`()
required: none

State

lastRet ∈ **D**(`Int`) **initially** 1

Actions

$u \twoheadrightarrow this$: `nextPrime`()
pre: true
state: no change

$u \leftarrow this$: `nextPrime` $\triangleleft\, p$
pre: $p >$ lastRet $\wedge\, isPrime(p) \wedge \forall i \in \{\text{lastRet} + 1, \ldots, p-1\} : \neg isPrime(i)$
state: lastRet $:= p$

Figure 13.8.: A specification of component automaton [`Sieve`](*this*) for [`Sieve`] component

$u \rightarrow$ head : `insert`(x) call. Because the call is targeted to the environment, the future u is stored in used.

[`Sieve`] **component specification.** Last but not least is the component specification $\mathcal{S}_{[\texttt{Sieve}]}$ (Figure 13.8). The specification states that every time a component instance returns a `nextPrime` call, the returned integer is the next lowest prime number. As a guide to determine the next prime number, the last returned integer is stored in the variable lastRet.

13.2.2 Class Verification

In this section, we sketch how the verification of the class implementation of the [`Sieve`] component can be done. First, we encode the αABS implementation of `Sieve` and `Filter` in SEQ (Listings 13.4 and 13.5). To shorten the encoding, we omit non-deterministic guessing of the resolved value of a future when the future is a placeholder of type `Unit`. The fields are collectively represented by the variable `fields`. From the encoding, we extract the weakest liberal precondition of each method in each class. These weakest liberal preconditions are shown to satisfy the class invariants extracted from the class specifications of `Sieve` (Figure 13.5) and `Filter` (Figure 13.6).

Listing 13.4: `Sieve` encoding in SEQ

```
IFilter head = null;
Int count = 2;
Fut<Int> retu = null;
trace t = [];
nextPrime(Fut<Int> fut) {
  Int next, next';
  Fut<Int> u;
  Int return;
  trace t1, t2, t3;

  t = t·fut ↠ this : nextPrime();
  assert I(fields, t) && wf(t);
  fields, t1 = some;
  t = t·t1;
  assume I(fields, t) && wf(t) &&
         head != null && retu == null;
  u = some;
  t = t·u → head : retrieve();
  retu = u;
  assume wf(t);
  assert I(fields, t) && wf(t);
  fields, t2, next' = some;
  t3 = retu ↞ head : retrieve ◁ next';
  next = next';
  t = t·t2·t3
  assume I(fields, t) && wf(t);
  retu = null;
  return = next;
  t = t·fut ← this : nextPrime ◁ return;
  assume wf(t);
}
```

```
generate(Fut<Unit> fut) {
  Fut<Unit> done, u';
  trace t', t'';

  t = t·fut ↠ this : generate();
  count = count + 1;
  u' = some;
  t = t·u' → head : check(count);
  done = u';
  assume wf(t);
  assert I(fields, t) && wf(t);
  fields, t' = some;
  t'' = done ↞ head : check ◁ Unit;
  t = t·t'·t'';
  assume I(fields, t) && wf(t);
  t = t·fut ← this : generate ◁ Unit;
  assume wf(t);
}

{
  Int h';
  Fut<Unit> u';

  h' = some;
  t = t·this → h' : new Filter(2,null);
  head = h';
  assume wf(t);
  u' = some;
  t = t·u' → this : generate();
  assume wf(t);
}
```

The method `nextPrime` of `Sieve` has the following weakest liberal precondition:

$$
\begin{aligned}
&wlp(\texttt{nextPrime}(fut), Q) = \\
&\quad \forall next, next', head', head'', retu', retu'', count', count'', u, t_1, t_2, \texttt{return} : \\
&\qquad next = next' = \texttt{return} \wedge retu'' = u \wedge \\
&\qquad (\mathbf{I}(\overline{f}, \mathrm{t}\cdot e_1) \wedge \mathbf{wf}(\mathrm{t}\cdot e_1)) \wedge \\
&\qquad (\mathbf{I}(\overline{f}', \mathrm{t}\cdot e_1\cdot t_1) \wedge \mathbf{wf}(\mathrm{t}\cdot e_1\cdot t_1) \wedge head' \neq \mathbf{null} \wedge retu' = \mathbf{null}) \implies \\
&\qquad\quad \mathbf{wf}(\mathrm{t}\cdot e_1\cdot t_1\cdot e_2) \implies \\
&\qquad\qquad (\mathbf{I}(\overline{f}', \mathrm{t}\cdot e_1\cdot t_1\cdot e_2) \wedge \mathbf{wf}(\mathrm{t}\cdot e_1\cdot t_1\cdot e_2)) \wedge \\
&\qquad\qquad (\mathbf{I}(\overline{f}'', \mathrm{t}\cdot e_1\cdot t_1\cdot e_2\cdot t_2\cdot e_3) \wedge \mathbf{wf}(\mathrm{t}\cdot e_1\cdot t_1\cdot e_2\cdot t_2\cdot e_3)) \implies \\
&\qquad\qquad\quad \mathbf{wf}(\mathrm{t}\cdot e_1\cdot t_1\cdot e_2\cdot t_2\cdot e_3\cdot e_4) \implies Q^{\overline{f}\cdot retu\cdot \mathrm{t}}_{\overline{f}''\cdot \mathbf{null}\cdot(\mathrm{t}\cdot e_1\cdot t_1\cdot e_2\cdot t_2\cdot e_3\cdot e_4)}
\end{aligned}
$$

Listing 13.5: `Filter` encoding in SEQ

```
// class parameters
Int mine;
Filter hd;
// fields
Filter next = null;
List<Int> todo = Nil;
List<Int> num = Cons(Nil, mine);
trace t = [];

check(Fut<Unit> fut, Int x) {
  t = t·fut ↠ this : check(x);
  todo = Cons(todo, x);
  t = t·fut ← this : check ◁ Unit;
  assume wf(t);
}

insert(Fut<Unit> fut, Int x) {
  t = t·fut ↠ this : insert(x);
  num = Cons(num, x);
  t = t·fut ← this : insert ◁ Unit;
  assume wf(t);
}

retrieve(Fut<Int> fut) {
  Int p, return;
  var t′;

  t = t·fut ↠ this : retrieve();
  assert I(fields, t) && wf(t);
  fields, t′ = some;
  t = t·t′;
  assume I(fields, t) && wf(t) &&
                          num != Nil;
  p = num.first();
  num = num.tail();
  return = p;
  t = t·fut ← this : retrieve ◁ return;
  assume wf(t);
}

{
  Fut<Unit> u';
  u' = some;
  t = t·u′ → this : filter();
  assume(t);
}
```

```
filter(Fut<Unit> fut) {
  Int x;
  Fut<Unit> u, u', u";
  Filter f';
  trace t1, t2, t3;
  t = t·fut ↠ this : filter();
  assert I(fields, t) && wf(t);
  fields, t1 = some; t = t·t1;
  assume I(fields, t) && wf(t) &&
                          todo != Nil;
  x = todo.first();
  if (x % mine != 0) {
    if (next == null) {
      if (hd == null) {
        f' = some;
    t = t·this → f' : new Filter(x,this);
        next = f'; assume wf(t);
        u' = some;
        t = t·u' → this : insert(x);
        u = u'; assume wf(t);
      }
      else {
        f' = some;
      t = t·this → f' : new Filter(x,hd);
        next = f'; assume wf(t);
        u' = some;
        t = t·u' → hd : insert(x);
        u = u'; assume wf(t);
      }
    } else {
      u' = some;
      t = t·u' → next : check(x);
      u = u'; assume wf(t);
    }
    assert I(fields, t) && wf(t);
    fields, t2 = some;
    t3 = u ↞ getTgt(t,u) : getMtd(t,u) ◁ Unit;
    t = t·t2·t3;
    assume I(fields, t) && wf(t);
  }
  todo = todo.tail();
  u" = some;
  t = t·u" → this : filter();
  assume wf(t);
  t = t·fut ← this : filter ◁ Unit;
  assume wf(t);
}
```

where the events e_1 are defined as follows:

- $e_1 = \langle fut \twoheadrightarrow this : \texttt{nextPrime}() \rangle$
- $e_2 = \langle u \rightarrow head' : \texttt{retrieve}() \rangle$
- $e_3 = \langle retu'' \twoheadrightarrow head'' : \texttt{retrieve} \triangleleft next \rangle$
- $e_4 = \langle fut \leftarrow this : \texttt{nextPrime} \triangleleft \texttt{return} \rangle$

For simplicity, the class fields are grouped together to a sequence of variables $\overline{f}$. The changes of values in the class fields are reflected by the different versions of the variables, e.g., $head'$ and $head''$ which represent the second and third versions of the variables $head'$ after the execution of the corresponding non-deterministic assignments. As explained in Section 11.2, the first version of these variables are supplied by the verification condition. Abusing the notation slightly, the assignment of *retu* back to **null** at the end of the method is reflected directly in the substitution on Q.

The method `generate` of `Sieve` has the following weakest liberal precondition:

$$\begin{array}{l} wlp(\texttt{generate}(fut), Q) = \\ \quad \forall done, u', head', t' : done = u' \wedge \\ \qquad \mathbf{wf}(\mathrm{t} \cdot e_1 \cdot e_2) \implies \\ \qquad\quad (\mathbf{I}(\overline{f}, \mathrm{t} \cdot e_1 \cdot e_2) \wedge \mathbf{wf}(\mathrm{t} \cdot e_1 \cdot e_2)) \wedge (\mathbf{I}(\overline{f}', \mathrm{t} \cdot e_1 \cdot e_2 \cdot t' \cdot e_3) \wedge \mathbf{wf}(\mathrm{t} \cdot e_1 \cdot e_2 \cdot t' \cdot e_3)) \implies \\ \qquad\qquad \mathbf{wf}(\mathrm{t} \cdot e_1 \cdot e_2 \cdot t' \cdot e_3 \cdot e_4) \implies Q^{\overline{f} \cdot \mathrm{t}}_{\overline{f}' \cdot (\mathrm{t} \cdot e_1 \cdot e_2 \cdot t' \cdot e_3 \cdot e_4)} \end{array}$$

where the events e_1, e_2, e_3 and e_4 are defined as follows:

- $e_1 = \langle fut \twoheadrightarrow \mathbf{this} : \texttt{generate}() \rangle$
- $e_2 = \langle done \rightarrow head : \texttt{check}(count + 1) \rangle$
- $e_3 = \langle done \twoheadleftarrow head' : \texttt{check} \triangleleft \mathbf{Unit} \rangle$
- $e_4 = \langle fut \leftarrow \mathbf{this} : \texttt{generate} \triangleleft \mathbf{Unit} \rangle$

The constructor of `Sieve` has the following weakest liberal precondition:

$$wlp(\texttt{Sieve}_{cons}, Q) = \forall h', u' : \mathbf{wf}(\mathrm{t} \cdot e_1) \implies \mathbf{wf}(\mathrm{t} \cdot e_1 \cdot e_2) \implies Q^{head \cdot \mathrm{t}}_{h' \cdot (\mathrm{t} \cdot e_1 \cdot e_2)}$$

where the events e_1 and e_2 are defined as follows:

- $e_1 = \langle \mathbf{this} \rightarrow h' : \mathbf{new}\ \texttt{Filter}(2, \mathbf{null}) \rangle$
- $e_2 = \langle u' \rightarrow \mathbf{this} : \texttt{generate}() \rangle$

The method `check` of `Filter` has the following weakest liberal precondition:

$$wlp(\texttt{check}(fut, x), Q) = \mathbf{wf}(\mathrm{t} \cdot e_1 \cdot e_2) \implies Q^{todo \cdot \mathrm{t}}_{\texttt{Cons}(x, todo) \cdot (\mathrm{t} \cdot e_1 \cdot e_2)}$$

where the events e_1 and e_2 are defined as follows:

- $e_1 = \langle fut \twoheadrightarrow \mathbf{this} : \texttt{check}(x) \rangle$
- $e_2 = \langle fut \leftarrow \mathbf{this} : \texttt{check} \triangleleft \mathbf{Unit} \rangle$

The weakest liberal precondition of the method `insert` of `Filter` is the same as that for `check` after substituting the method name `check` with `insert` and the field variable *todo* with *num*.

The method `retrieve` of `Filter` has the following weakest liberal precondition:

$$
\begin{aligned}
&wlp(\texttt{retrieve}(fut), Q) = \\
&\quad \forall num', \texttt{return}, t' : \texttt{return} = num'.\texttt{first()} \wedge \\
&\qquad (\mathbf{I}(\overline{f}, \mathrm{t} \cdot e_1) \wedge \mathbf{wf}(\mathrm{t} \cdot e_1)) \wedge (\mathbf{I}(\overline{f}', \mathrm{t} \cdot e_1 \cdot t') \wedge \mathbf{wf}(\mathrm{t} \cdot e_1 \cdot t') \wedge num' \neq \texttt{Nil}) \implies \\
&\qquad\quad \mathbf{wf}(\mathrm{t} \cdot e_1 \cdot t' \cdot e_2) \implies Q^{\overline{f} \cdot \mathrm{t}}_{\overline{f}' \cdot (\mathrm{t}' \cdot e_1 \cdot t' \cdot e_2)}
\end{aligned}
$$

where the events e_1 and e_2 are defined as follows:

- $e_1 = \langle fut \twoheadrightarrow \mathbf{this} : \texttt{retrieve}() \rangle$
- $e_2 = \langle fut \twoheadleftarrow \mathbf{this} : \texttt{retrieve} \triangleleft \texttt{return} \rangle$

Figure 13.9 presents the weakest liberal precondition of method `filter` of `Filter`, where the events e_1, e_2, e_2', e_3, e_3', e_3'', e_4, e_4', e_4'', e_5 and e_6 are defined as follows:

- $e_1 = \langle fut \twoheadrightarrow \mathbf{this} : \texttt{filter}() \rangle$
- $e_2 = \langle \mathbf{this} \rightarrow f' : \mathbf{new}\ \texttt{Filter}(x, \mathbf{this}) \rangle$
- $e_2' = \langle \mathbf{this} \rightarrow f' : \mathbf{new}\ \texttt{Filter}(x, hd) \rangle$
- $e_3 = \langle u' \rightarrow \mathbf{this} : \texttt{insert}(x) \rangle$
- $e_3' = \langle u' \rightarrow hd : \texttt{insert}(x) \rangle$
- $e_3'' = \langle u' \rightarrow next' : \texttt{check}(x) \rangle$
- $e_4 = \langle u' \twoheadleftarrow \mathbf{this} : \texttt{insert} \triangleleft \mathbf{Unit} \rangle$
- $e_4' = \langle u' \twoheadleftarrow hd : \texttt{insert} \triangleleft \mathbf{Unit} \rangle$
- $e_4'' = \langle u' \twoheadleftarrow next' : \texttt{check} \triangleleft \mathbf{Unit} \rangle$
- $e_5 = \langle u'' \rightarrow \mathbf{this} : \texttt{filter}() \rangle$
- $e_6 = \langle fut \twoheadleftarrow \mathbf{this} : \texttt{filter} \triangleleft \mathbf{Unit} \rangle$

This precondition is more complex because of the presence of the branching statements which produce 4 branches. These branches are produced hierarchically based on three factors:

- the divisibility of the first number on the *todo* list by the number held by the `Filter` actor,
- whether the `Filter` actor is the last actor in the chain, and
- whether the `Filter` actor is the head actor of the chain.

The constructor of `Filter` has the following weakest liberal precondition:

$$
wlp(\texttt{Filter}_{cons}, Q) = \forall u' : \mathbf{wf}(\mathrm{t} \cdot u' \rightarrow \mathbf{this} : \texttt{filter}()) \implies Q^{\mathrm{t}}_{\mathrm{t} \cdot u' \rightarrow \mathbf{this}:\texttt{filter}()}
$$

To obtain class invariants from the class specifications $\mathcal{S}_{\texttt{Sieve}}$ and $\mathcal{S}_{\texttt{Filter}}$, we define a predicate ρ for each class that connects the class attributes with the internal variables present in the specification. For the `Sieve` class, the values of

$$
\begin{aligned}
&wlp(\texttt{filter}(fut), Q) = \\
&\quad \forall todo', todo'', todo''', next', f', u, u', u'', x, t_1, t_2 : \\
&\qquad next' = f' \wedge u = u' \wedge x = todo'.\texttt{first()} \wedge \\
&\qquad (mine \mid x \implies todo''' = todo''.\texttt{tail()}) \wedge \\
&\qquad (mine \nmid x \implies todo'' = todo'.\texttt{tail()}) \wedge \\
&\qquad (\mathbf{I}(\overline{f}, \mathsf{t} \cdot e_1) \wedge \mathbf{wf}(\mathsf{t} \cdot e_1)) \wedge (\mathbf{I}(\overline{f}', \mathsf{t} \cdot e_1 \cdot t_1) \wedge \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot t_1) \wedge todo' \neq \texttt{Nil}) \implies \\
&\qquad\quad ((mine \mid x \wedge next' = \mathbf{null} \wedge hd = \mathbf{null}) \wedge \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot t_1 \cdot e_2) \implies \\
&\qquad\qquad \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot t_1 \cdot e_2 \cdot e_3) \implies \\
&\qquad\qquad\quad (\mathbf{I}(\overline{f}', \mathsf{t} \cdot e_1 \cdot t_1 \cdot e_2 \cdot e_3) \wedge \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot t_1 \cdot e_2 \cdot e_3)) \wedge \\
&\qquad\qquad\quad (\mathbf{I}(\overline{f}'', \mathsf{t} \cdot e_1 \cdot t_1 \cdot e_2 \cdot e_3 \cdot t_2 \cdot e_4) \wedge \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot t_1 \cdot e_2 \cdot e_3 \cdot t_2 \cdot e_4) \implies \\
&\qquad\qquad\qquad \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot t_1 \cdot e_2 \cdot e_3 \cdot t_2 \cdot e_4 \cdot e_5) \implies \\
&\qquad\qquad\qquad\quad \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot t_1 \cdot e_2 \cdot e_3 \cdot t_2 \cdot e_4 \cdot e_5 \cdot e_6) \implies Q^{\overline{f} \cdot \mathsf{t}}_{\overline{f}''' \cdot (\mathsf{t} \cdot e_1 \cdot t_1 \cdot e_2 \cdot e_3 \cdot t_2 \cdot e_4 \cdot e_5 \cdot e_6)}) \\
&\qquad\quad \vee \\
&\qquad\quad ((mine \mid x \wedge next' = \mathbf{null} \wedge hd \neq \mathbf{null}) \wedge \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot t_1 \cdot e_2') \implies \\
&\qquad\qquad \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot t_1 \cdot e_2' \cdot e_3') \implies \\
&\qquad\qquad\quad (\mathbf{I}(\overline{f}', \mathsf{t} \cdot e_1 \cdot t_1 \cdot e_2' \cdot e_3') \wedge \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot t_1 \cdot e_2' \cdot e_3')) \wedge \\
&\qquad\qquad\quad (\mathbf{I}(\overline{f}'', \mathsf{t} \cdot e_1 \cdot t_1 \cdot e_2' \cdot e_3' \cdot t_2 \cdot e_4') \wedge \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot t_1 \cdot e_2' \cdot e_3' \cdot t_2 \cdot e_4') \implies \\
&\qquad\qquad\qquad \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot t_1 \cdot e_2' \cdot e_3' \cdot t_2 \cdot e_4' \cdot e_5) \implies \\
&\qquad\qquad\qquad\quad \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot t_1 \cdot e_2' \cdot e_3' \cdot t_2 \cdot e_4' \cdot e_5 \cdot e_6) \implies Q^{\overline{f} \cdot \mathsf{t}}_{\overline{f}''' \cdot (\mathsf{t} \cdot e_1 \cdot t_1 \cdot e_2 \cdot e_3' \cdot t_2 \cdot e_4' \cdot e_5 \cdot e_6)}) \\
&\qquad\quad \vee \\
&\qquad\quad ((mine \mid x \wedge next' \neq \mathbf{null}) \wedge \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot t_1 \cdot e_3'') \implies \\
&\qquad\qquad (\mathbf{I}(\overline{f}', \mathsf{t} \cdot e_1 \cdot t_1 \cdot e_3'') \wedge \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot t_1 \cdot e_3'')) \wedge \\
&\qquad\qquad (\mathbf{I}(\overline{f}'', \mathsf{t} \cdot e_1 \cdot t_1 \cdot e_3'' \cdot t_2 \cdot e_4'') \wedge \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot t_1 \cdot e_3'' \cdot t_2 \cdot e_4'') \implies \\
&\qquad\qquad\quad \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot t_1 \cdot e_3'' \cdot t_2 \cdot e_4'' \cdot e_5) \implies \\
&\qquad\qquad\qquad \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot t_1 \cdot e_3'' \cdot t_2 \cdot e_4'' \cdot e_5 \cdot e_6) \implies Q^{\overline{f} \cdot \mathsf{t}}_{\overline{f}''' \cdot (\mathsf{t} \cdot e_1 \cdot t_1 \cdot e_3'' \cdot t_2 \cdot e_4'' \cdot e_5 \cdot e_6)}) \\
&\qquad\quad \vee \\
&\qquad\quad (mine \nmid x \wedge \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot t_1 \cdot e_5) \implies \mathbf{wf}(\mathsf{t} \cdot e_1 \cdot t_1 \cdot e_5 \cdot e_6) \implies Q^{\overline{f} \cdot \mathsf{t}}_{\overline{f}'' \cdot (\mathsf{t} \cdot e_1 \cdot t_1 \cdot e_5 \cdot e_6)})
\end{aligned}
$$

Figure 13.9.: The weakest liberal precondition for method `filter` of class `Filter`

head, *count*, *retu*, and *done* must be the same as the values of head, c, retu, and tmpFut respectively.

$$\rho_{\texttt{Sieve}}(\overline{f}, s) \stackrel{\text{def}}{=} head = s(\underline{\text{head}}) \wedge count = s(\underline{\text{c}}) \wedge retu = s(\underline{\text{retu}}) \wedge done = \underline{\text{tmpFut}}$$

For the `Filter` class, we equate the class attributes *mine*, *hd*, *next*, *todo*, *num* to

the internal variables myDiv, head, next, todo and divs, respectively.

$$\rho_{\texttt{Filter}}(\overline{f}, s) \stackrel{\text{def}}{=} \begin{array}{l} mine = s(\underline{\text{myDiv}}) \wedge hd = s(\underline{\text{head}}) \wedge next = s(\underline{\text{next}}) \wedge \\ todo = s(\underline{\text{todo}}) \wedge num = s(\underline{\text{divs}}) \end{array}$$

Verifying whether a class invariant holds for each method is established by identifying the corresponding event sequence transitions every time the class invariant is checked, as demonstrated in the previous section. To verify the method `nextPrime` of `Sieve`, we consider the first three event sequence transition specifications of Figure 13.5. What needs to be noted from the other transition specifications is that the value of head is changed only once by the constructor, and thus remains constant afterwards. The three transition specifications do not change the value of c and tmpFut, so we only need to check at the completion of the method, whether the return value and the value of the field *retu* correspond to the specification. The return value corresponds to whatever value the `Sieve` actor gets from retrieving from the head, and the value of *retu* is set back to **`null`**, equalling the specification. Thus, the invariant holds for `nextPrime`. Verifying the method `generate` and the constructor is more straightforward.

Verifying `Filter` implementation for `insert`, `check`, `retrieve` and the constructor is relatively straightforward. Verifying the proof obligation of `filter` is more involved, because of multiple branching and **`await`** statements. The crux of the verification effort is the linkage between events after each **`await`** statement. Because the specification $\mathcal{S}_{\texttt{Filter}}$ strictly restricts that a filtering process can progress only at the correct stages and the futures of the `insert` and `check` calls are stored temporarily, a proper relation between, e.g., `insert` emittance call events e_3 and their reaction e_4 is maintained.

Let us consider the case when the first number on the `todo` list is not divisible by the prime number held by the `Filter` actor, and the actor is the first in the chain. From the weakest liberal precondition of `filter`, we can assume that the class invariant holds at every release point prior to the last one. That is, we proceed under the assumption that given an execution where the trace $t \cdot e_1 \cdot t_1 \cdot e_2 \cdot e_3 \cdot t_2 \cdot e_4$ is introduced by executing the appropriate event sequence transitions. For the method `filter` to continue its execution, the `insert` method must finish executing. That is, the last transition specification $\mathcal{S}_{\texttt{Filter}}$ must be applicable. After executing this transition, we arrive at a release point and have to check that the predicate $\rho_{\texttt{Filter}}$ holds. Because this transition only changes the value of todo by removing its first number, and the value of *todo* in the weakest liberal precondition is updated in the same way, the predicate is evaluated to true. Other cases are reasoned similarly.

13.2.3 System Verification

The next task is to verify that the implementation of the sieve of Eratosthenes satisfies its specification. Using the class specifications $\mathcal{S}_{\texttt{Sieve}}$ and $\mathcal{S}_{\texttt{Filter}}$, we proceed with this task in two stages. First we show that the component specification $\mathcal{S}_{[\texttt{Filter}]}$ is satisfied. Using this component specification, we show that the component specification $\mathcal{S}_{[\texttt{Sieve}]}$ is satisfied. In both instances, we make use of some properties of the reachable states of the implementation.

Verifying [`Filter`] component implementation. From the class specification $\mathcal{S}_{\texttt{Filter}}$, it is clear that integers are added to a `Filter` actor in the order of the respective reaction events to `check` and `insert` calls are performed. That is, the todo and divs lists capture the received integers, such that none are lost unless they are properly retrieved or filtered. To show that $\mathcal{S}_{[\texttt{Filter}]}$ is satisfied, we need to ensure that the `Filter` actor performs the filtering process one integer at a time. By using a combination of trace projection and regular expression, we can formulate the property as done by the following lemma.

Lemma 13.1 (Filter sequentiality):
Let $\mathcal{A}_{\texttt{Filter}}$ be the actor automaton specified by $\mathcal{S}_{\texttt{Filter}}$ and $t \in \mathit{Traces}(\mathcal{A}_{\texttt{Filter}})$. Then,

$$t{\downarrow_{\texttt{filter}}}\ \textbf{pr}\ \_ \rightarrow \mathit{this} : \texttt{filter}() \cdot (\_ \twoheadrightarrow \mathit{this} : \texttt{filter}() \cdot \_ \rightarrow \mathit{this} : \texttt{filter}() \cdot \_ \leftarrow \mathit{this} : \texttt{filter} \triangleleft \texttt{Unit})^*$$

As defined in Section 4.3, the projection to a method name is a shorthand for projecting to a set of call and return events where the respective method of these events are the given method name. The property above states that a trace of a `Filter` actor with respect to the `filter` method is a prefix of a trace that starts with a single `filter` emittance call event, followed by a cycle of reaction event, followed by a call emittance event and a return emittance event, all bearing the `filter` method name. Note that `filter` is an internal method. Therefore, only a self call can trigger the execution of the method. An underscore _ represents that we can put any value in it. In this case, we left out the exact values of the futures. Because the traces of an actor automaton are well-formed, the futures that we left out from the regular expression are interconnected such that the futures of a call reaction event and a return emittance event are the future of the preceding call emittance event. The proof for this property follows from how the states of $\mathcal{A}_{\texttt{Filter}}$ evolve as specified in $\mathcal{S}_{\texttt{Filter}}$.

Proof (Induction on the length of the corresponding execution):
Let $\alpha \in \mathit{execs}(\mathcal{A}_{\texttt{Filter}})$ be an execution such that $t = \mathit{trace}(\alpha)$ and the execution ends in state s. If this state s is an initial state (i.e., t is an empty trace), the property is trivially satisfied.

For the inductive case, assume that the trace t satisfies the property. From the event sequence transition specifications of $\mathcal{S}_{\texttt{Sieve}}$, we need to show that the following holds.

- $\underline{\texttt{fStg}} = 1$ if $t{\downarrow}\texttt{filter}$ is either empty or ends with a `filter` call emittance or return emittance event; and
- $\underline{\texttt{fStg}} \in \{2,3\}$ if $t{\downarrow}\texttt{filter}$ ends with a `filter` call reaction event.

If the cycle of the filtering stage holds, the property holds. We proceed by analyzing a transition $s \xrightarrow{e} s'$ for each possible event e.

Case $e = \langle u \to \mathit{this} : \texttt{filter}() \rangle$: Either $s(\underline{\texttt{fStg}}) = 1 \wedge \neg s(\mathit{cons})$ or $s(\underline{\texttt{fStg}}) \in \{2,3\}$. Otherwise, the transition cannot occur. Following the observation on the last event of the projected trace $t{\downarrow}\texttt{filter}$, appending e to this projected trace does not break the property. In the former case, the class constructor is executed such that by the semantics of class specifications (Definition 10.4) $s'(\mathit{cons}) = \mathit{true}$, disallowing the class constructor to be executed again. In the latter case, executing this transition changes the value of $\underline{\texttt{fStg}}$ to 1. Therefore, the observation on the value of $\underline{\texttt{fStg}}$ holds.

Case $e = \langle u \leftarrow \mathit{this} : \texttt{filter} \triangleleft \texttt{Unit} \rangle$: This transition can only occur if $s(\underline{\texttt{fStg}}) = 1 \wedge \neg s(\mathit{release}) \wedge t{\downarrow}\texttt{filter}$ ends with $u' \to \mathit{this} : \texttt{filter}()$. The actor is not at a release point because it still needs to complete an event sequence transition. From $\mathcal{S}_{\texttt{Filter}}$ and ignoring the input events, the event preceding e can only be a `filter` call emittance event. Following this observation, the property is satisfied.

Case $e = \langle u \twoheadrightarrow \mathit{this} : \texttt{filter}() \rangle$: This case occurs when $\mathit{emitOf}(e) \in s(\mathit{buf}_c)$. From the induction hypothesis, each `filter` call is consumed immediately after its emittance, except for the last one. Therefore, the call buffer in s contains exactly one such call event. $t{\downarrow}\texttt{filter} \cdot e$ satisfies the property, and because $s'(\underline{\texttt{fStg}} = 2)$, the observation on $\underline{\texttt{fStg}}$ holds.

Case $e = \langle \_ \to \_ : \texttt{new Filter}(\_, \_) \rangle$: The transition changes the value of $\underline{\texttt{fStg}}$ to 3, but because it does not add any event to the projected trace, the observation on $\underline{\texttt{fStg}}$ still holds.

Case $e = \langle \_ \to s(\underline{\texttt{next}}) : \texttt{check}(\_) \rangle$: Same as above.

Other cases: the value of $\underline{\texttt{fStg}}$ does not change and $t \cdot e{\downarrow}\texttt{filter} = t{\downarrow}\texttt{filter}$.

$$
\begin{array}{l}
\textsc{PreReq} \wedge \forall x \in rStates(\mathcal{C}_1), y \in states(\mathcal{C}_2) : \exists \mathbb{C}_1, \mathbb{C}_2, \mathcal{A}_1, \mathcal{A}_2, s_1, s_2 : \\
\quad \mathbb{C}_1 = config(\mathcal{C}_1)(x) \wedge \mathbb{C}_2 = config(\mathcal{C}_2)(y) \wedge \\
\quad (\mathcal{A}_1, s_1, false) \in \mathbb{C}_1 \wedge (\mathcal{A}_2, s_2, true) \in \mathbb{C}_2 \wedge \{f\} = names(\mathcal{A}_1) \wedge f \in names(\mathcal{A}_2) \wedge \\
\quad s_2(known) = \{a \mid f \in ancestors(a) \vee (s_1(\underline{\mathsf{head}}) \neq \texttt{null} \wedge a = s_1(\underline{\mathsf{head}}))\} \wedge \\
\quad s_2(\underline{\mathsf{myDiv}}) = s_1(\underline{\mathsf{myDiv}}) \wedge s_2(\underline{\mathsf{head}}) = s_1(\underline{\mathsf{head}}) \wedge \\
\quad (s_1(\underline{\mathsf{next}}) = \texttt{null} \implies \\
\qquad s_2(\underline{\mathsf{kept}}) = sift(s_1(\underline{\mathsf{todo}}), \{s_1(\underline{\mathsf{myDiv}})\}) \wedge \\
\qquad s_2(\underline{\mathsf{divs}}) = select(s_1(\underline{\mathsf{divs}}), s_1(\underline{\mathsf{insIdx}})) \wedge \\
\qquad s_2(\underline{\mathsf{sifted}}) = siftSeq(s_1(\underline{\mathsf{todo}}), s_1(\underline{\mathsf{myDiv}})) \wedge \\
\qquad s_2(outCalls) = s_1(outCalls) \wedge s_2(\overline{genFut}) = s_2(\underline{\mathsf{used}}) = s_1(\underline{\mathsf{used}})) \wedge \\
\quad (s_1(\underline{\mathsf{next}}) \neq \texttt{null} \implies \\
\qquad \exists \mathcal{A}, s : (\mathcal{A}, s, true) \in \mathbb{C}_1 \wedge s_1(\underline{\mathsf{next}}) \in names(\mathcal{A}) \wedge \\
\qquad\quad s_2(\underline{\mathsf{kept}}) = sift(s_1(\underline{\mathsf{todo}}), \{s_1(\underline{\mathsf{myDiv}})\} \cup s(\underline{\mathsf{kept}})) \wedge \\
\qquad\quad s_2(\underline{\mathsf{divs}}) = select(s_1(\underline{\mathsf{divs}}), s_1(\underline{\mathsf{insIdx}})) \wedge \\
\qquad\quad s_2(\underline{\mathsf{sifted}}) = siftSeq(s_1(\underline{\mathsf{todo}}), \{s_1(\underline{\mathsf{myDiv}})\} \cdot s(\underline{\mathsf{sifted}})) \wedge \\
\qquad\quad s_2(outCalls) = (s_1(outCalls) \cup s(\overline{outCalls})) \cap eCore(extEv(f)) \wedge \\
\qquad\quad (s_1(\underline{\mathsf{head}}) = \texttt{null} \implies s_2(genFut) = \emptyset) \wedge \\
\qquad\quad (s_1(\underline{\mathsf{head}}) \neq \texttt{null} \implies s_2(genFut) = s_2(\underline{\mathsf{used}}) = s_1(\underline{\mathsf{used}}) \cup s(\underline{\mathsf{used}}))) \\
\quad \implies r(x) = y
\end{array}
$$

Figure 13.10.: A possibility map from $\mathcal{C}_1$ representing `Filter` implementation to $\mathcal{C}_2$ representing $\mathcal{S}_{[\texttt{Filter}]}$

Each event transition extends the trace t such that the property holds. Thus by the induction argument, the property holds. □

Figure 13.10 defines a possibility map r from the CCA representation of the `Filter` implementation to the CCA representing the component specification $\mathcal{S}_{[\texttt{Filter}]}$. The map r from reachable states x of $\mathcal{C}_1$ to states y of $\mathcal{C}_2$ fulfills the following conditions:

- The prerequisite PreReq is satisfied.
- The class parameters of the initial `Filter` actor f in $\mathcal{C}_1$ and $\mathcal{C}_2$ hold the same values.
- If f is the last actor in the chain ($\underline{\mathsf{next}} = \texttt{null}$), the resulting filtering process represented by the component instance is collected exclusively from the $\underline{\mathsf{todo}}$ list, the class parameter $\underline{\mathsf{myDiv}}$ and the list of divisors $\underline{\mathsf{divs}}$ of the `Filter` actor. The filtering process is represented by the *sift* function which returns the set

of sifted integers (or *siftSeq* function if the order is relevant). This function is defined as follows:

$$\begin{aligned} sift([\,], result) &= result \\ sift(i \cdot s, result) &= \begin{cases} sift(s, result \cup \{i\}) & \text{if } \forall j \in result : j \nmid i \\ sift(s, result) & \text{otherwise} \end{cases} \end{aligned}$$

The collected list of divisors divs for the component instance that are obtained through the `insert` calls is extracted by projecting the list of divs of the `Filter` actor to the list of indices insldx. The projection is performed by the function *select* defined below.

$$\begin{aligned} select(s, [\,]) &= [\,] \\ select(s, i \cdot s') &= s[i] \cdot select(s, s') \end{aligned}$$

The remaining state variables *outCalls* and *genFut* of the component automaton $\mathcal{A}_{[\texttt{Filter}]}$ are assigned to the value of the variable *outCalls* held by the actor automaton $\mathcal{A}_{\texttt{Filter}}$ and the internal variable used, respectively.

- If f is not the last actor in the chain (next $\neq$ `null`), the states of component automaton $\mathcal{A}_{[\texttt{Filter}]}$ are collected from the states of the `Filter` actor and the created [`Filter`] subcomponent.

Now we show that r is a possibility map, by showing that r is indeed a map, and the two conditions in Definition 12.4 hold.

0. For each reachable state $x \in rStates(\mathcal{C}_1)$, there exists exactly one state where $y \in states(\mathcal{C}_2)$. This property holds because all state variables of the CompA specified by $\mathcal{S}_{[\texttt{Filter}]}$ are assigned to values stored by the variables of the representation of the [`Filter`] implementation. Thus, r is a map.

1. All initial states of $\mathcal{C}_1$ are mapped to some initial state of $\mathcal{C}_2$, as the class parameters have to be the same and the set of known actors on the component side is fixed to all component actors plus the `Filter` actor provided as a class parameter.

2. For each event transition $x \xrightarrow{e} x'$ executable by $\mathcal{C}_1$ in some reachable state x, we show that the mapping condition specified in Definition 12.4.2 holds. For simplicity, we assume f is the initial `Filter` actor, f' represents the next filter in the chain, and head represents the class parameter `hd` stored by the AA in $\mathcal{C}_1$ representing f and the corresponding CompA in $\mathcal{C}_2$. Because none of the methods return a `Filter` actor, only the initial actor is the exposed actor.

Case $e = \langle u \twoheadrightarrow f : \texttt{check}(i) \rangle$: Reacting to this input call means adding the integer i to the todo list. Because this event is an observable generated event of the [Filter] component instance, a transition with the same label must be executable at $r(x)$ and the post-state is $r(x')$. From $\mathcal{S}_{[\texttt{Filter}]}$, this transition can be executed at $r(x)$ and accordingly the post-state is $r(x')$.

Case $e = \langle u \twoheadrightarrow f : \texttt{insert}(i) \rangle$: Similar to above, except that the concerned list is divs and the insIdx.

Case $e = \langle u \twoheadleftarrow f : mtd \triangleleft \texttt{Unit} \rangle$ **where** $mtd \in \{\texttt{check}, \texttt{insert}\}$: By the semantics of class specifications (Definition 10.4), all events but the first event in the transition sequence do not change the internal state. As this is mimicked by event transition specifications of the respective return events in $\mathcal{S}_{[\texttt{Filter}]}$, the property holds for this case.

Case $e = \langle u \twoheadrightarrow f : \texttt{retrieve}() \rangle$: Same as above.

Case $e = \langle u \twoheadleftarrow f : \texttt{retrieve} \triangleleft i \rangle$: The return event removes the head of the list divs and decreases the insIdx list. Because the component specification allows a choice whether to reduce the divs or sifted lists of the component specification, we take the transition whenever possible where the head of the sifted list is returned, unless the first element of insIdx is 1. Thus, $r(x) \xrightarrow{\mathcal{C}_2} r(x')$.

Case $e = \langle u \twoheadrightarrow f : \texttt{filter}() \rangle$: This event is not part of the observable generated event of the [Filter] component instance. As the execution of this event only changes the value of fStg, and this variable does not play any part in the map, $r(x) = r(x')$.

Case $e = \langle u \rightarrow f : \texttt{filter}() \rangle$: This event is not part of the observable generated event of the [Filter] component instance. When this event is executed, the todo list is either already reduced or the first element, which is divisible by myDiv, is removed from the list. Thus, nothing changes from the kept and sifted lists and $r(x) = r(x')$.

Case $e = \langle u \leftarrow f : \texttt{filter}() \rangle$: Similar to above, except that in no circumstances is the todo list changed.

Case $e = \langle f \rightarrow f' : \texttt{new Filter}(i, f'') \rangle$: This event is not part of the observable generated event of the [Filter] component instance. When transition is executed, a new instance of the [Filter] component automaton is added to the configuration of x'. Nothing changes for the variables kept and sifted because the integer i is already recognized as a

divisor. Therefore, $r(x) = r(x')$.

Case $e = \langle u \to \underline{\text{head}} : \texttt{insert}(i)\rangle$: This event is part of the observable generated event of the [Filter] component instance. Its execution adds u to the used set. Thus, $r(x) \xrightarrow{\mathcal{C}_2} r(x')$.

Case $e = \langle u \twoheadleftarrow \underline{\text{head}} : \texttt{insert} \triangleleft \texttt{Unit}\rangle$: This event is part of the observable generated event of the [Filter] component instance. Its execution removes the first element for the todo list. Because the removed integer is sent to head, which is an actor not part of the component instance, this integer is removed from the list sifted as described in $\mathcal{S}_{[\texttt{Filter}]}$. Thus, $r(x) \xrightarrow{\mathcal{C}_2} r(x')$.

Case $e = \langle u \to f : \texttt{insert}(i)\rangle$: This event is not part of the observable generated event of the [Filter] component instance. When the transition is executed, only the values of fStg, used and done are changed. As they play no role in the map, $r(x) = r(x')$.

Case $e = \langle u \twoheadleftarrow f : \texttt{insert} \triangleleft \texttt{Unit}\rangle$: This event is not part of the observable generated event of the [Filter] component instance. Executing this transition removes the first element of the todo list. However, the removed element has become part of the subcomponent instance. Therefore, $r(x) = r(x')$.

Case $e = \langle u \to \underline{\text{next}} : \texttt{check}(i)\rangle$: This event is not part of the observable generated event of the [Filter] component instance. When the transition is executed, only the values of fStg and done are changed. As they play no role in the map, $r(x) = r(x')$.

Case $e = \langle u \twoheadleftarrow \underline{\text{next}} : \texttt{check} \triangleleft \texttt{Unit}\rangle$: Similar to the case for the reaction event to insert.

As the conditions hold, r is a possibility map and the Filter class implementation satisfies $\mathcal{S}_{[\texttt{Filter}]}$.

If feed the [Filter] component instance correctly, it will generate prime numbers, as the following lemma shows.

Lemma 13.2 (Prime number generations):
Assume the following:

- $\mathcal{A}$ be a component automaton specified by $\mathcal{S}_{[\texttt{Filter}]}$ component instance,
- s_0 be an initial state of $\mathcal{A}$ such that $s_0(\underline{\text{myDiv}}) = 2$,
- α is an execution of $\mathcal{A}$ starting from s_0, and
- $t \in \mathit{trace}(\alpha)$, such that $\mathit{gatherTodo}(t{\downarrow}\texttt{check})$ **pr** $3{\cdot}4{\cdot}5{\cdot}\ldots$ where $\mathit{gatherTodo}$

Lemma 13.2 (Continued)
is a function that extracts the sequence of integer parameters from `check` call reaction events, and $t{\downarrow}\texttt{insert} = []$.
Then,

$$gatherRet(t{\downarrow}\texttt{retrieve})\ \textbf{pr}\ 2 \cdot 3 \cdot 5 \cdot \ldots$$

where *gatherRet* is a function that extracts the sequence of integer returns from `retrieve` return emittance events.

Proof:
Follows from $\mathcal{S}_{[\texttt{Filter}]}$ and proof on the Sieve of Eratosthenes algorithm (e.g., [LP93]). □

Verifying [`Sieve`] component implementation. The implementation of the `Sieve` class and the [`Filter`] component have been verified. They become the basis of verifying the implementation of the [`Sieve`] component.

First we note several properties about the reachable states. The `Sieve` class provides a specific usage context of the [`Filter`] component, such that the `insert` method of the [`Filter`] component is never called. Thus, divs in [`Filter`] is always an empty sequence. The return value of a `retrieve` call depends solely on the integers the component instance receives. If we can show that the `Sieve` actor creates a [`Filter`] component instance with the first divisor 2 and supplies it with an increasing sequence of consecutive integers starting from 3, the component instance produces a sequence of consecutive prime numbers starting from 2.

Similar to a `Filter` actor, a `Sieve` actor forms a recursive call to provide the [`Filter`] component instance with the desired sequence. We can reproduce the result of Lemma 13.1 in terms of `generate` and `check` method calls as given below.

Lemma 13.3 (Generator sequentiality):
Let $\mathcal{A}_{\texttt{Sieve}}$ be the actor automaton specified by $\mathcal{S}_{\texttt{Sieve}}$ and $t \in Traces(\mathcal{A}_{\texttt{Sieve}})$.
Then,

$$t{\downarrow}\{\texttt{generate}, \texttt{check}\}\ \textbf{pr}\ \_ \rightarrow this : \texttt{generate}() \cdot (\_ \twoheadrightarrow this : \texttt{generate}() \cdot \_ \rightarrow \underline{\text{head}} : \texttt{check}(\_) \cdot \_ \leftarrow \underline{\text{head}} : \texttt{check} \triangleleft \textbf{Unit} \cdot \_ \twoheadleftarrow \underline{\text{head}} : \texttt{check} \triangleleft \textbf{Unit} \cdot \_ \rightarrow this : \texttt{generate}() \cdot \_ \leftarrow this : \texttt{generate} \triangleleft \textbf{Unit})^*$$

The proof follows the proof for Lemma 13.1, except that there is only one release point introduced when generating the integers.

Proof (by induction on the length of the corresponding execution):
Let $\alpha \in execs(\mathcal{A}_{\texttt{Filter}})$ be an execution such that $t = trace(\alpha)$ and the execution ends in state s. If this state s is an initial state (i.e., t is an empty trace), the

property is trivially satisfied.

For the inductive case, assume that the trace t satisfies the property. From the event sequence transition specifications of $\mathcal{S}_{\texttt{Sieve}}$, we observe that

- $\underline{\texttt{tmpFut}} \neq \texttt{null}$, if the trace $t{\downarrow}\{\texttt{generate}, \texttt{check}\}$ ends with a reaction to a `generate` self-call or a `check` return from the `Filter` head actor.
- $\underline{\texttt{tmpFut}} = \texttt{null}$ otherwise.

We proceed by analyzing each kind of event sequence transition $s \xrightarrow{\bar{e}} s'$ such that the `Sieve` actor is at a release point ($s(\mathit{release}) = \mathit{true}$).

Case $\bar{e} = \langle \mathit{this} \to \underline{\texttt{head}} : \texttt{new Filter}(2, \texttt{null})\rangle \cdot \langle u \to \mathit{this} : \texttt{generate}()\rangle$: This is the only event sequence transition that creates a `Filter` actor and produces a `generate` call without having a precondition. However, representing a class constructor, this event sequence transition can only occur once. Before this transition occurs, $\underline{\texttt{head}}$ is `null` and $\underline{\texttt{tmpFut}} = \texttt{null}$ disabling the execution of other event sequence transitions involving `generate` and `check` calls. Therefore, the projected trace after appending $\bar{e}$ contains the first `generate` call emittance event.

Case $\bar{e} = \langle u \twoheadrightarrow \mathit{this} : \texttt{generate}()\rangle \cdot \langle u' \to \underline{\texttt{head}}.\texttt{check}(\underline{\texttt{c}})\rangle$: This event sequence transition occurs only when $\underline{\texttt{head}} \neq \texttt{null}$ and the corresponding call event exists. By the induction hypothesis and the above case analysis, when this transition is executed, the automaton is a state where the class constructor has been executed. The property on the trace t holds after appending $\bar{e}$.

Case $\bar{e} = \langle u \twoheadleftarrow \underline{\texttt{head}}.\texttt{check} \triangleleft \texttt{Unit}\rangle \cdot \langle u' \to \mathit{this} : \texttt{generate}()\rangle \cdot \langle u'' \leftarrow \mathit{this} : \texttt{generate} \triangleleft \texttt{Unit}\rangle$:
Before this event sequence transition can occur, the corresponding return emittance event must appear in the trace. An AA produces only well-formed traces (Lemma 7.1), so this emittance event must be preceded by a corresponding call event. As the trace t satisfies the property, appending it with the events $\bar{e}$ still satisfies the property. The transition causes the actor to forget the future u, so the same reaction event cannot be used in a transition.

Other cases: The other event sequence transitions do not involve `generate` and `check` calls and do not change the internal state needed for checking the preconditions of the three event sequence transitions above.

Each event sequence transition extends the trace t such that the property holds. Thus by the induction argument, the property holds. □

The lemma above means that the [`Filter`] component instance created by a `Sieve` actor receives all integers in the right order.

Corollary 13.1:
Let $\mathcal{C}$ be a component configuration automaton whose initial actor is of the class `Sieve`, and $t \in \mathit{Traces}(\mathcal{C})$. Then,

$$\mathit{gatherTodo}(t{\downarrow}\texttt{check}) \textbf{ pr } 3 \cdot 4 \cdot 5 \cdot \ldots$$

Proof:
Follows from $\mathcal{S}_{\texttt{Sieve}}$, $\mathcal{S}_{[\texttt{Filter}]}$, the well-formedness of traces of CCA (Corollary 9.2) and Lemma 13.3. □

The final piece we need to verify the implementation of [`Sieve`] is a possibility map from the reachable states of the CCA representing the implementation to the states of the CCA representing the component specification. The main idea of the map is to correctly assign the last returned prime number variable lastRet of the component specification to the corresponding variable on the class specification $\mathcal{S}_{\texttt{Sieve}}$. The class specification $\mathcal{S}_{\texttt{Sieve}}$ has variables prev and tmpPrev that store the last returned prime number. Being temporary, we use tmpPrev as lastRet only if the `Sieve` actor is in the middle of executing the event sequence transition that returns the `nextPrime` call. Otherwise, the value of lastRet always points to prev. Assuming that the initial `Sieve` actor has the name *sv*, the following map r provides the formal definition of the desired map.

$$\begin{aligned}
&\textsc{PreReq} \wedge \forall x \in \mathit{rStates}(\mathcal{C}_1), y \in \mathit{states}(\mathcal{C}_2) : \exists \mathbb{C}_1, \mathbb{C}_2, \mathcal{A}_1, \mathcal{A}_2, s_1, s_2 : \\
&\quad \mathbb{C}_1 = \mathit{config}(\mathcal{C}_1)(x) \wedge \mathbb{C}_2 = \mathit{config}(\mathcal{C}_2)(y) \wedge \\
&\quad (\mathcal{A}_1, s_1, \mathit{false}) \in \mathbb{C}_1 \wedge (\mathcal{A}_2, s_2, \mathit{true}) \in \mathbb{C}_2 \wedge \{sv\} = \mathit{names}(\mathcal{A}_1) \wedge sv \in \mathit{names}(\mathcal{A}_2) \wedge \\
&\quad s_2(\mathit{known}) = \{a \mid sv \in \mathit{ancestors}(a)\} \wedge s_2(\mathit{outCalls}) = \emptyset \wedge \\
&\quad (\neg s_1(\mathit{release}) \wedge s_1(\underline{\mathsf{tmpPrev}}) \neq 0 \implies s_2(\underline{\mathsf{lastRet}}) = s_1(\underline{\mathsf{tmpPrev}})) \wedge \\
&\quad (s_1(\mathit{release}) \vee s_1(\underline{\mathsf{tmpPrev}}) = 0 \implies s_2(\underline{\mathsf{lastRet}}) = s_1(\underline{\mathsf{prev}})) \\
&\quad \implies r(x) = y
\end{aligned}$$

We show that r is a possibility map, by showing that r is indeed a map, and the two conditions in Definition 12.4 hold.

0. For each reachable state $x \in \mathit{rStates}(\mathcal{C}_1)$, there exists exactly one state where $y \in \mathit{states}(\mathcal{C}_2)$. This follows directly from the definition of r.

1. There is only a single initial state in $\mathcal{C}_1$ and $\mathcal{C}_2$, as there are no class parameters and the initial value of the internal variables are fixed. Because the values of prev and lastRet on the initial configurations are the same, the condition Definition 12.4.1 holds.

2. From Lemma 13.2 and corollary 13.1, we know that for each reachable state of $\mathcal{C}_1$, each `retrieve` call made by the `Sieve` actor to the filter chain returns the next smallest prime number. Now we prove that the condition Definition 12.4.2 on the transitions hold. That is, if x is a reachable state of $\mathcal{C}_1$ and $x \xrightarrow{e}_{\mathcal{C}_1} x'$ is a transition, we need to show if e is an observable generated event of a component instance of [`Sieve`], then $r(x) \xrightarrow{e}_{\mathcal{C}_2} r(x')$, otherwise $r(x) = r(x')$.

   Case $e = \langle u \twoheadrightarrow \mathit{this} : \texttt{nextPrime}() \rangle$: This event is an observable generated event of [`Sieve`]. When this transition is executed, according to $\mathcal{S}_{\texttt{Sieve}}$, the `Sieve` actor moves from one release point to another. According to $\mathcal{S}_{[\texttt{Sieve}]}$, executing this does not affect the internal state of the component instance. Thus, lastRet stays equal to prev and $r(x) \xrightarrow{e}_{\mathcal{C}_2} r(x')$.

   Case $e = \langle u \twoheadleftarrow \mathit{this} : \texttt{retrieve} \triangleleft i \rangle$: This event is not an observable generated event of [`Sieve`]. When this transition is executed, according to $\mathcal{S}_{\texttt{Sieve}}$, the `Sieve` actor assigns tmpPrev to the (old) value of prev, and the value of prev is updated to i. Because the `Sieve` actor does not reach a release point, the value of tmpPrev is used in the mapping instead of prev. Thus, $r(x) = r(x')$.

   Case $e = \langle u \twoheadleftarrow \mathit{this} : \texttt{nextPrime} \triangleleft i \rangle$: This event is an observable generated event of [`Sieve`]. When this transition is executed, the `Sieve` actor reaches a release point, and the value of lastRet is mapped to prev $= i$. By Corollary 13.1, we know that i is indeed the smallest prime number larger than tmpPrev. Then, $r(x) \xrightarrow{e}_{\mathcal{C}_2} r(x')$.

   Other cases: The event is not an observable generated event of [`Sieve`]. When this event is generated by the filter chain, the mapping is not affected. When the `Sieve` actor in x is at a release point, according to $\mathcal{S}_{\texttt{Sieve}}$, the value of tmpPrev becomes assigned to 0. Consequently, the value of lastRet is mapped to prev, maintaining the condition. When the `Sieve` actor in x is not at a release point, then lastRet $= 0$ due to $\mathcal{S}_{\texttt{Sieve}}$ and x being a reachable state. Therefore, the map condition holds.

Therefore, we have shown that the implementation Listing 13.3 fulfills the specification $\mathcal{S}_{[\texttt{Sieve}]}$.

13.3 Discussion

The ticker factory example shows that to come up with a possibility map, class specifications may need to be enriched by extra information that may not play a part in verifying the implementation. Relaxing the condition of the possibility map to a normal simulation relation, where a state of the implementation may be mapped to more than one state of the CCA of the specification, would ease the description. However, this would require an appropriate definition of the relation for it to imply trace inclusion.

The sieve example provides a rather elaborate implementation in αABS. The main source of the complexity is the lack of FIFO guarantee in processing incoming messages. This FIFO guarantee is simulated in the sieve case by forming a recursive call that only progresses once the call is fully completed. This approach is also applicable in general to simulate loops. The inclusion of the FIFO guarantee or other scheduling mechanisms can be accommodated in the automaton model by placing a more specific data structure to represent the buffer and apply appropriate constraints to simulate the desired scheduling mechanism. Without such a scheduling mechanism, it is difficult to guarantee, for example, that a prime number that has been returned by the filter chain is actually returned by the sieve.

The sieve example also uncovers some complexity in the class specification technique. When it is desired that certain internal variables are assigned new values only after the execution of an event in an event sequence transition other than the first event, we must introduce temporary internal variables and some usage convention. Allowing more flexibility to which intermediary state the value of a certain variable should change requires a stronger connection between the AA and the class invariants than what is currently provided by Lemma 10.1.

The way the [`Filter`] component is used by the `Sieve` class suggests that the specification of the filter component could actually be simplified. A [`Filter`] component instance never receives an `insert` call from its user. Therefore, the variable divs in $\mathcal{S}_{[\texttt{Filter}]}$ is not needed when we want to verify the behavior of the [`Sieve`] component. However, because components are specified with an open context, all possible incoming calls must be taken into account. This aspect enables reuse of the [`Filter`] component (and its specification) in other situations.

CHAPTER 14

Conclusion

The actor model is an established model to develop distributed systems, with strong research interests being poured to further investigate various aspects related to the actor model. The goal of this thesis is to provide a compositional verification technique to reason about implementations of open actor systems that feature non-shared futures and cooperative multitasking. To reach this goal, we assembled various modeling and reasoning techniques within an automaton framework. The following sections recount the contributions in more detail and describe the outlook.

14.1 Contributions

To achieve the goal, we developed an adaptation of the DIOA model to represent the behavior of actor systems in an open setting. The model captures the creation of new actors and the modification of communication capabilities of the actors based on information it receives from other actors, particularly the actors they are exposed to. By precisely characterizing the actions actors can perform and storing information about the exposed actors, we resolved the question of how to represent an object-based setting using the DIOA model. Furthermore, we incorporated into the model the concept of non-shared futures, which improves the decoupling of the process that sends a computation request to an actor and the process that retrieves the result. This adaptation also becomes the basis for a specification technique that allows the use of both states and traces.

The automaton framework provides the basis to realize a two-tier verification method. First the implementation is verified against the class specifications given as automata. Then, the behavior of the system, also specified as an automaton, is hierarchically verified by means of the class specifications.

As the basis of the implementation, we presented a class-based actor language αABS, derived from ABS: a melting pot of design languages, minimal executable formalisms and implementation languages fit for representing distributed systems. This language features non-shared futures to communicate results of calls

on actors and cooperative multitasking which allows actors to process multiple calls concurrently. We established a sound connection between αABS and the adapted DIOA model through the use of trace-based class invariants. This connection is formed by bringing together

- a trace-based denotational semantics of αABS,
- a sound transformation of αABS into a simple sequential language SEQ with a weakest liberal precondition semantics, and
- a translation of constructive specifications of SIOA into class invariants.

We introduced a notion of components to allow compositional reasoning based on the class specifications. The component notion incorporates the idea of an activator class and actor creation tree such that

- a component can be statically referred to just by the activator class, and
- an instance of the component is captured by grouping together all actors transitively created by the head actor.

The classes needed to implement a component can be uniquely derived from the activator class. As a result, the components have a hierarchical structure ideal for a compositional reasoning.

To verify the component implementations, we lifted the possibility map for I/O automata to the DIOA setting. This specialized simulation relation provides a state-based means to conclude if the traces produced by the implementation of a component are part of the allowed traces of its specification. As a system can also be represented by some activator class that acts as the initializer for the system, we have a compositional means to verify whether a system implementation satisfies desired system properties.

14.2 Outlook

We envision several directions for further research. One direction is to feature shared futures and loops as part of the automaton model. This model would allow full support of the ABS language. A possible approach to represent shared futures is to define a parameterized SIOA that represents their behavior. Because all futures have the same behavior (resolved once then fetched multiple times), it is sufficient to define this parameterized SIOA once and have them as invariants in the verification. Care is needed when dealing with futures that are passed by the environment to the system as parameters, but the generation of which occurs in the environment.

Another direction is to explore how interesting properties of AA, CompA, ACA and CCA can be specified and verified. Logics are good means to serve this pur-

pose. A lead on how this may be done is present on a preliminary effort to apply Lamport's TLA on the DIOA model [Kap09]. An integration of the temporal logic could also be fruitful. It is also interesting to see how the DIOA model is affected by the various bisimulation and simulation relations widely investigated for process calculi.

Georgiou et al. [GLMT09] propose an automated implementation of distributed algorithms from I/O automata. This synthesization idea is also present in ABS, where ABS programs are automatically translated into popular implementation languages such as Java, Scala and Haskell. Developing such a synthesizing mechanism for AA focuses the verification effort only on the second tier: proving that a system property holds from the class specifications. Because AA generally are unimplementable, an important question to be addressed is what kind of AA is implementable. An open question is also present in the other direction. Given a class implementation, it is currently unknown whether an AA that represents this class implementation exists. The way αABS can be transformed to SEQ seems to indicate that this is indeed the case. From such an extraction, further methods could be developed to synthesize readable class specifications, replacing the need to come up with class specifications in the first place. This extracted version can also act as the baseline for enriching class specifications with information that may be required for proving that the implementation satisfies component specifications.

Bibliography

[AAKR14] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmet Kara, and Othmane Rezine. "Verification of Dynamic Register Automata". In: *FSTTCS 2014*. To be published. 2014 (cit. on pp. 114, 115).

[Abr93] Samson Abramsky. "Computational Interpretations of Linear Logic". In: *Theor. Comput. Sci.* 111.1&2 (1993), pp. 3–57 (cit. on p. 156).

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996, pp. I–XIII, 1–396 (cit. on p. 62).

[AD12] Wolfgang Ahrendt and Maximilian Dylla. "A System for Compositional Verification of Asynchronous Objects". In: *Sci. Comput. Program.* 77.12 (2012), pp. 1289–1309 (cit. on pp. 10, 41, 42, 50, 52, 59, 156, 170, 181, 249).

[AD90] Rajeev Alur and David L. Dill. "Automata For Modeling Real-Time Systems". In: *ICALP*. 1990, pp. 322–335 (cit. on p. 116).

[AD99] Thomas Arts and Mads Dam. "Verifying a Distributed Database Lookup Manager Written in Erlang". In: *World Congress on Formal Methods*. 1999, pp. 682–700 (cit. on pp. 17, 20, 181).

[AGGS09] Erika Ábrahám, Immo Grabe, Andreas Grüner, and Martin Steffen. "Behavioral interface description of an object-oriented language with futures and promises". In: *J. Log. Algebr. Program.* 78.7 (2009), pp. 491–518 (cit. on pp. 7, 30).

[Agh86] Gul Abdulnabi Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986 (cit. on pp. 1, 4, 38, 39, 60, 63).

[AH01] Luca de Alfaro and Thomas A. Henzinger. "Interface automata". In: *ESEC / SIGSOFT FSE*. 2001, pp. 109–120 (cit. on pp. 7, 79).

[AL01] Paul Camille Attie and Nancy Ann Lynch. "Dynamic Input/Output Automata: A Formal Model for Dynamic Systems". In: *CONCUR*. 2001, pp. 137–151 (cit. on pp. 7, 75, 114).

[AL15] Paul Camille Attie and Nancy Ann Lynch. “Dynamic Input/Output Automata: A Formal and Compositional Model for Dynamic Systems”. In: *Information and Computation* (2015). To appear (cit. on pp. xvii, 7, 8, 75–77, 79, 82, 86, 87, 114, 143, 180).

[AMST97] Gul Abdulnabi Agha, Ian Alistair Mason, Scott Fraser Smith, and Carolyn L. Talcott. “A Foundation for Actor Computation”. In: *J. Funct. Program.* 7.1 (1997), pp. 1–72 (cit. on pp. 6, 47, 64, 65, 121).

[Apt81] Krzysztof Rafał Apt. “Ten Years of Hoare’s Logic: A Survey - Part 1”. In: *ACM Trans. Program. Lang. Syst.* 3.4 (1981), pp. 431–483 (cit. on pp. 24, 159, 160).

[Apt84] Krzysztof Rafał Apt. “Ten Years of Hoare’s Logic: A Survey Part II: Nondeterminism”. In: *Theor. Comput. Sci.* 28 (1984), pp. 83–109 (cit. on pp. 10, 24, 25, 159, 160).

[Arb04] Farhad Arbab. “Reo: a channel-based coordination model for component composition”. In: *Mathematical Structures in Computer Science* 14.3 (2004), pp. 329–366 (cit. on p. 117).

[Arm03] Joe Armstrong. “Making Reliable Distributed Systems in the Presence of Software Errors”. PhD thesis. Royal Institute of Technology, Stockholm, Sweden, 2003 (cit. on pp. 1, 38).

[Arm10] Joe Armstrong. “Erlang”. In: *Commun. ACM* 53 (9 2010), pp. 68–75 (cit. on pp. 4, 38).

[AT04] Gul Agha and Prasanna Thati. “An Algebraic Theory of Actors and Its Application to a Simple Object-Based Language”. In: *Essays in Memory of Ole-Johan Dahl*. 2004, pp. 26–57 (cit. on pp. 62, 181).

[AØVWW13] Taslim Arif, Bjarte M. Østvold, Karina Villela, Balthasar Weitzel, and Peter Wong. *ABS Tool Platform and Methodology*. Deliverable 1.5 of project FP7-231620 (HATS). 2013 (cit. on pp. 5, 29).

[BCHKS13] Benedikt Bollig, Aiswarya Cyriac, Loïc Hélouët, Ahmet Kara, and Thomas Schwentick. “Dynamic Communicating Automata and Branching High-Level MSCs”. In: *LATA*. 2013, pp. 177–189 (cit. on pp. 114, 115).

[BCJ07] Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. “A Complete Guide to the Future”. In: *ESOP*. 2007, pp. 316–330 (cit. on pp. 38, 170).

[BH10] Benedikt Bollig and Loïc Hélouët. “Realizability of Dynamic MSC Languages”. In: *CSR 2010, Kazan, Russia*. 2010, pp. 48–59 (cit. on pp. 114, 115).

[BH77] Henry Givens Baker Jr. and Carl Hewitt. “The Incremental Garbage Collection of Processes”. In: *SIGART Bull.* (64 1977), pp. 55–59 (cit. on pp. 5, 18, 30).

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of object-oriented software: The KeY approach*. Berlin, Heidelberg: Springer, 2007 (cit. on p. 170).

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008 (cit. on pp. 2, 21, 78, 137).

[BKWZ13] Kshitij Bansal, Eric Koskinen, Thomas Wies, and Damien Zufferey. “Structural Counter Abstraction”. In: *TACAS*. 2013, pp. 62–77 (cit. on p. 182).

[Boe02] Frank S. de Boer. “A Hoare Logic for Dynamic Networks of Asynchronously Communicating Deterministic Processes”. In: *Theor. Comput. Sci.* 274.1–2 (2002), pp. 3–41 (cit. on p. 181).

[Bro02] Stephen D. Brookes. “Traces, Pomsets, Fairness and Full Abstraction for Communicating Processes”. In: *CONCUR*. 2002, pp. 466–482 (cit. on p. 41).

[BSAR06] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan J. M. M. Rutten. “Modeling component connectors in Reo by constraint automata”. In: *Sci. Comput. Program.* 61.2 (2006), pp. 75–113 (cit. on p. 117).

[BVBF13] Abdeldjalil Boudjadar, Frits Willem Vaandrager, Jean-Paul Bodeveix, and Mamoun Filali. “Extending UPPAAL for the Modeling and Verification of Dynamic Real-Time Systems”. In: *FSEN*. 2013, pp. 111–132 (cit. on pp. 114, 116).

[BZ83] Daniel Brand and Pitro Zafiropulo. “On Communicating Finite-State Machines”. In: *J. ACM* 30.2 (1983), pp. 323–342 (cit. on pp. 7, 114, 115).

[Büc60] Julius Richard Büchi. “On a decision method in restricted second order arithmetic”. In: *International Congress on Logic, Methodoly and philosophy of Science*. Stanford University Press, 1960, pp. 1–11 (cit. on p. 182).

[CG97] Luca Cardelli and Andrew D. Gordon. "Mobile Ambients". In: *Electr. Notes Theor. Comput. Sci.* 10 (1997), pp. 198–201 (cit. on p. 6).

[CH05] Denis Caromel and Ludovic Henrio. *A theory of distributed objects - asynchrony, mobility, groups, components*. Springer, 2005, pp. I–XXXII, 1–346 (cit. on p. 2).

[CHS04] Denis Caromel, Ludovic Henrio, and Bernard P. Serpette. "Asynchronous and deterministic objects". In: *POPL*. 2004, pp. 123–134 (cit. on p. 62).

[CHS09] Denis Caromel, Ludovic Henrio, and Bernard P. Serpette. "Asynchronous sequential processes". In: *Inf. Comput.* 207.4 (2009), pp. 459–495 (cit. on pp. 39, 62).

[Cla+07] Manuel Clavel, Francisco Duràn, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, eds. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Vol. 4350. LNCS. Springer, 2007 (cit. on p. 5).

[Cli81] William Douglas Clinger. "Foundations of Actor Semantics". PhD thesis. Cambridge, MA: Massachusetts Institute of Technology, 1981 (cit. on pp. 41, 61).

[CNW13] Dave Clarke, James Noble, and Tobias Wrigstad, eds. *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Vol. 7850. LNCS. Springer, 2013 (cit. on pp. 64, 72).

[CY83] Bo-Shoe Chen and Raymond T. Yeh. "Formal Specification and Verification of Distributed Systems". In: *IEEE Trans. Software Eng.* 9.6 (1983), pp. 710–722 (cit. on p. 2).

[DCDMY09] Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Dimitris Mostrous, and Nobuko Yoshida. "Objects and Session Types". In: *Inf. Comput.* 207.5 (2009), pp. 595–641 (cit. on p. 156).

[DDJO12] Crystal Chang Din, Johan Dovland, Einar Broch Johnsen, and Olaf Owe. "Observable Behavior of Distributed Systems: Component Reasoning for Concurrent Objects". In: *J. Log. Algebr. Program.* 81.3 (2012), pp. 227–256 (cit. on pp. 10, 11, 22, 24, 26, 43, 59, 95, 154, 155, 159, 160, 170, 172, 181).

[DDO12a] Crystal Chang Din, Johan Dovland, and Olaf Owe. "An Approach to Compositional Reasoning about Concurrent Objects and Futures". In: *Research Report 415* (2012) (cit. on pp. 10, 11, 24, 26, 170).

[DDO12b] Crystal Chang Din, Johan Dovland, and Olaf Owe. "Compositional Reasoning about Shared Futures". In: *SEFM*. 2012, pp. 94–108 (cit. on p. 170).

[DFG97] Mads Dam, Lars-Åke Fredlund, and Dilian Gurov. "Toward Parametric Verification of Open Distributed Systems". In: *COMPOS*. 1997, pp. 150–185 (cit. on pp. 156, 181).

[DG94] John Darlington and Yike Guo. "Formalising Actors in Linear Logic". In: *OOIS*. 1994, pp. 37–53 (cit. on p. 156).

[Dij75] Edsger Wybe Dijkstra. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". In: *Commun. ACM* 18.8 (1975), pp. 453–457 (cit. on pp. 10, 24).

[Dij76] Edsger Wybe Dijkstra. *A Discipline of Programming*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1976 (cit. on pp. 159, 164).

[DJO05] Johan Dovland, Einar Broch Johnsen, and Olaf Owe. "Verification of Concurrent Objects with Asynchronous Method Calls". In: *SwSTE*. 2005, pp. 141–150 (cit. on pp. 10, 11, 24, 154, 160).

[DJO08] Johan Dovland, Einar Broch Johnsen, and Olaf Owe. "Observable Behavior of Dynamic Systems: Component Reasoning for Concurrent Objects". In: *ENTCS* 203.3 (2008), pp. 19–34 (cit. on pp. 59, 154).

[DKOne] Emanuele D'Osualdo, Jonathan Kochems, and Chih-Hao Luke Ong. "Automatic Verification of Erlang-Style Concurrency". In: *SAS*. 2013, pp. 454–476 (cit. on pp. 72, 143, 182).

[DN66] Ole-Johan Dahl and Kristen Nygaard. "SIMULA - an ALGOL-based simulation language". In: *Commun. ACM* 9.9 (1966), pp. 671–678 (cit. on p. 4).

[DOB14] Crystal Chang Din, Olaf Owe, and Richard Bubel. "Runtime Assertion Checking and Theorem Proving for Concurrent and Distributed Systems". In: *MODELSWARD*. 2014, pp. 480–487 (cit. on p. 170).

[Dua99] Carlos Henrique Cabral Duarte. “Proof-Theoretic Foundations for the Design of Actor Systems”. In: *Mathematical Structures in Computer Science* 9.3 (1999), pp. 227–252 (cit. on pp. 156, 181).

[ECM11] European Computer Manufacturers Association. *Standard ECMA-262: ECMAScript Language Specification*. Tech. rep. Geneva: ECMA International, 2011 (cit. on p. 4).

[EH86] E. Allen Emerson and Joseph Y. Halpern. ““Sometimes” and “Not Never” Revisited: On Branching Versus Linear Time Temporal Logic”. In: *J. ACM* 33.1 (Jan. 1986), pp. 151–178 (cit. on p. 156).

[Fin90] Alain Finkel. “Reduction and covering of infinite reachability trees”. In: *Inf. Comput.* 89.2 (1990), pp. 144–179 (cit. on p. 182).

[Fis+11] Jasmin Fisher, Thomas A. Henzinger, Dejan Nickovic, Nir Piterman, Anmol V. Singh, and Moshe Y. Vardi. “Dynamic Reactive Modules”. In: *CONCUR*. 2011, pp. 404–418 (cit. on pp. 114, 116).

[FL05] Marco Faella and Axel Legay. “Some Models and Tools for Open Systems”. In: *Foundations of Interface Technologies*. 2005 (cit. on pp. 7, 75).

[FS01] Alain Finkel and Phillipe Schnoebelen. “Well-structured transition systems everywhere!” In: *Theor. Comput. Sci.* 256.1-2 (2001), pp. 63–92 (cit. on p. 182).

[FS07] Lars-Åke Fredlund and Hans Svensson. “McErlang: a model checker for a distributed functional programming language”. In: *SIGPLAN Not.* 42.9 (Oct. 2007), pp. 125–136 (cit. on p. 182).

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995 (cit. on pp. 5, 31, 115).

[GJJ96] James Gosling, William N. Joy, and Guy L. Steele Jr. *The Java Language Specification*. Addison-Wesley, 1996 (cit. on p. 5).

[GLMT09] Chryssis Georgiou, Nancy A. Lynch, Panayiotis Mavrommatis, and Joshua A. Tauber. “Automated implementation of complex distributed algorithms specified in the IOA language”. In: *STTT* 11.2 (2009), pp. 153–171 (cit. on pp. 143, 221).

[Gre75] Irene Greif. “Semantics of Communicating Parallel Processes”. PhD thesis. Cambridge, MA, USA: Massachusetts Institute of Technology, 1975 (cit. on p. 61).

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997 (cit. on p. 13).

[GZ99] Mauro Gaspari and Gianluigi Zavattaro. "An Algebra of Actors". In: *FMOODS*. 1999 (cit. on pp. 62, 181).

[Hal85] Robert H. Halstead Jr. "Multilisp: A Language for Concurrent Symbolic Computation". In: *ACM Trans. Program. Lang. Syst.* 7.4 (1985), pp. 501–538 (cit. on pp. 5, 18, 30, 39).

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence". In: *IJCAI*. 1973, pp. 235–245 (cit. on pp. 1, 4, 38, 39).

[Hen88] Matthew Hennessy. *Algebraic theory of processes*. MIT Press series in the foundations of computing. MIT Press, 1988, pp. I–VI, 1–270 (cit. on p. 181).

[HO09] Philipp Haller and Martin Odersky. "Scala Actors: Unifying thread-based and event-based programming". In: *Theor. Comput. Sci.* 410.2-3 (2009), pp. 202–220 (cit. on pp. 5, 30).

[Hoa78] Charles Antony Richard Hoare. "Communicating Sequential Processes". In: *Commun. ACM* 21.8 (Aug. 1978), pp. 666–677 (cit. on pp. 5, 6, 62).

[HP85] David Harel and Amir Pnueli. "Logics and Models of Concurrent Systems". In: ed. by Krzysztof Rafał Apt. New York, NY, USA: Springer, 1985. Chap. On the Development of Reactive Systems, pp. 477–498 (cit. on p. 65).

[HT91] Kohei Honda and Mario Tokoro. "An Object Calculus for Asynchronous Communication". In: *ECOOP*. 1991, pp. 133–147 (cit. on p. 62).

[HTK00] David Harel, Jerzy Tiuryn, and Dexter Kozen. *Dynamic Logic*. Cambridge, MA, USA: MIT Press, 2000 (cit. on pp. 155, 170).

[Häh+10] Reiner Hähnle, Einar Broch Johnsen, Bjarte M. Østvold, Jan Schäfer, Martin Steffen, and Arild B. Torjusen. *Report on the Core ABS Language and Methodology: Part A*. Part of Deliverable 1.1 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`. 2010 (cit. on pp. 31, 35, 243).

[Häh13] Reiner Hähnle. "The Abstract Behavioral Specification Language: A Tutorial Introduction". In: *International School on Formal Models for Components and Objects: Post Proceedings*. Vol. 7688. LNCS. Springer, 2013, pp. 1–37 (cit. on pp. 31, 40).

[IT11] International Telecommunication Union – Telecommunication Standardization. *Recommendation Z.120: Message Sequence Chart (MSC)*. Tech. rep. Geneva: ISO/IEC, 2011 (cit. on p. 115).

[IT95] International Telecommunication Union – Telecommunication Standardization. *Open Distributed Processing – Reference Models parts 1–4*. Tech. rep. Geneva: ISO/IEC, 1995 (cit. on pp. 1, 4, 29).

[JC10] Mohammad Mahdi Jaghoori and Tom Chothia. "Timed Automata Semantics for Analyzing Creol". In: *FOCLASA*. 2010, pp. 108–122 (cit. on p. 117).

[JHSSS11] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. "ABS: A Core Language for Abstract Behavioral Specification". In: *FMCO 2010*. LNCS. Graz, Austria: Springer, 2011, pp. 142–164 (cit. on pp. 4, 5, 18, 29, 31, 35, 38, 170).

[JO04] Einar Broch Johnsen and Olaf Owe. "Object-Oriented Specification and Open Distributed Systems". In: *Essays in Memory of Ole-Johan Dahl*. 2004, pp. 137–164 (cit. on p. 59).

[Jon03] Simon Peyton Jones. *Haskell 98 language and libraries : the revised report*. Cambridge U.K.; New York: Cambridge University Press, 2003 (cit. on p. 5).

[JOY06] Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. "Creol: A type-safe object-oriented model for distributed concurrent systems". In: *Theor. Comput. Sci.* 365.1-2 (2006), pp. 23–66 (cit. on pp. 4, 29, 38, 39, 117, 170).

[JR05] Alan Jeffrey and Julian Rathke. "Java Jr: Fully Abstract Trace Semantics for a Core Java Language". In: *ESOP*. 2005, pp. 423–438 (cit. on p. 7).

[Kap09] Tatjana Kapus. "Using Mobile TLA as a Logic for Dynamic I/O Automata". In: *IEICE Transactions on Information and Systems* E92.D.8 (2009), pp. 1515–1522 (cit. on p. 221).

[KBR05] Marcel Kyas, Frank S. de Boer, and Willem P. de Roever. "A Compositional Trace Logic for Behavioural Interface Specifications". In: *Nord. J. Comput.* 12.2 (2005), pp. 116–132 (cit. on p. 59).

[KF94] Michael Kaminski and Nissim Francez. “Finite-Memory Automata”. In: *Theor. Comput. Sci.* 134.2 (1994), pp. 329–363 (cit. on pp. 114, 115).

[KGV13] Christian Krause, Holger Giese, and Erik Peter de Vink. “Compositional and behavior-preserving reconfiguration of component connectors in Reo”. In: *J. Vis. Lang. Comput.* 24.3 (2013), pp. 153–168 (cit. on p. 117).

[KPH12] Ilham W. Kurnia and Arnd Poetzsch-Heffter. “A Relational Trace Logic for Simple Hierarchical Actor-Based Component Systems”. In: AGERE! ’12. Tucson, Arizona, USA: ACM, 2012, pp. 47–58 (cit. on pp. 180, 253).

[KPH13] Ilham W. Kurnia and Arnd Poetzsch-Heffter. “Verification of Open Concurrent Object Systems”. In: *FMCO 2012*. Vol. 7866. LNCS. Springer, 2013, pp. 83–118 (cit. on pp. 5, 12, 72, 253).

[KPH15] Ilham W. Kurnia and Arnd Poetzsch-Heffter. *A Dynamic Automaton Model for Open Actor-Based Systems*. Tech. rep. University of Kaiserslautern, 2015 (cit. on pp. 12, 254).

[KPHW10] Ilham W. Kurnia, Arnd Poetzsch-Heffter, and Yannick Welsch. “State-based Object Models Are More Abstract Than Trace-based Models: Towards a Unified Specification Framework”. In: *Technical Report No. 2010-13*. 2010-13. Karlsruhe, 2010, pp. 268–282 (cit. on p. 254).

[Kro85] Stein Krogdahl. “Multiple Inheritance in SIMULA-like Languages”. In: *BIT* 25.2 (1985), pp. 318–326 (cit. on p. 40).

[Lam83a] Leslie Lamport. “Specifying Concurrent Program Modules”. In: *ACM Trans. Program. Lang. Syst.* 5.2 (1983), pp. 190–222 (cit. on pp. 20, 147, 155).

[Lam83b] Leslie Lamport. “What Good is Temporal Logic?” In: *IFIP Congress*. 1983, pp. 657–668 (cit. on p. 6).

[LBR06] Gary Todd Leavens, Albert Louis Baker, and Clyde Ruby. “Preliminary Design of JML: A Behavioral Interface Specification Language for Java”. In: *ACM SIGSOFT Software Engineering Notes* 31.3 (2006), pp. 1–38 (cit. on p. 29).

[LC06] Gary Todd Leavens and Yoonsik Cheon. *Design by Contract with JML*. 2006 (cit. on p. 5).

[Leo90] John Leo. “Dynamic Process Creation in a Static Model”. MA thesis. MIT, 1990 (cit. on p. 7).

[Lie87] Henry Lieberman. “Object-oriented Concurrent Programming”. In: ed. by Akinori Yonezawa and Mario Tokoro. Cambridge, MA, USA: MIT Press, 1987. Chap. Concurrent Object-oriented Programming in Act 1, pp. 9–36 (cit. on pp. 4, 38).

[LMWF93] Nancy Ann Lynch, Michael Merritt, William E. Weihl, and Alan Fekete. *Atomic Transactions*. Morgan Kaufmann, 1993 (cit. on p. 7).

[LP93] François Leclerc and Christine Paulin-Mohring. “Programming with Streams in Coq - A Case Study: the Sieve of Eratosthenes”. In: *TYPES*. 1993, pp. 191–212 (cit. on p. 214).

[LS88] Barbara Liskov and Liuba Shrira. “Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems”. In: *PLDI*. 1988, pp. 260–267 (cit. on pp. 5, 18, 30, 39).

[LS95] Robert Gregory Lavender and Douglas Craig Schmidt. “Active Object – An Object Behavioral Pattern for Concurrent Programming”. In: *Pattern Languages of Programs*. 1995 (cit. on p. 40).

[LT87] Nancy Ann Lynch and Mark Rogers Tuttle. “Hierarchical Correctness Proofs for Distributed Algorithms”. In: *PODC*. 1987, pp. 137–151 (cit. on pp. 7, 10, 27, 75, 171, 174).

[LV95] Nancy A. Lynch and Frits W. Vaandrager. “Forward and Backward Simulations: I. Untimed Systems”. In: *Inf. Comput.* 121.2 (1995), pp. 214–233 (cit. on p. 149).

[Lyn96] Nancy Ann Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996 (cit. on pp. 2, 75, 78, 95, 103, 125, 143).

[Man01] Panagiotis Manolios. “Mechanical Verification of Reactive Systems”. PhD thesis. University of Texas at Austin, 2001 (cit. on p. 174).

[Maz86] Antoni W. Mazurkiewicz. “Trace Theory”. In: *Petri Nets*. 1986, pp. 279–324 (cit. on p. 60).

[MC81] Jayadev Misra and Kanianthra Mani Chandy. “Proofs of Networks of Processes”. In: *IEEE Trans. Software Eng.* 7.4 (1981), pp. 417–426 (cit. on pp. 3, 159).

[Mey08] Roland Meyer. “On Boundedness in Depth in the π-Calculus”. In: *IFIP TCS*. 2008, pp. 477–489 (cit. on p. 182).

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1997 (cit. on p. 5).

[Mic99] Microsoft. *Component Object Model*. Jan. 1999 (cit. on pp. 5, 71).

[Mil82] Robin Milner. *A Calculus of Communicating Systems*. Secaucus, NJ, USA: Springer, 1982 (cit. on p. 62).

[Mil99] Robin Milner. *Communicating and Mobile Systems – The π-Calculus*. Cambridge University Press, 1999, pp. I–XII, 1–161 (cit. on pp. 5, 6, 29, 62).

[Mor87] Joseph M. Morris. "Varieties of Weakest Liberal Preconditions". In: *Inf. Process. Lett.* 25.3 (1987), pp. 207–210 (cit. on p. 165).

[NS94] Tobias Nipkow and Konrad Slind. "I/O Automata in Isabelle/HOL". In: *TYPES*. 1994, pp. 101–119 (cit. on pp. xviii, 10, 27, 171, 174, 175).

[NS95] Rocco De Nicola and Roberto Segala. "A process algebraic view of input/output automata". In: *TCS* 138.2 (1995). Meeting on the mathematical foundation of programing semantics, pp. 391–423 (cit. on p. 154).

[OA88] Ernst-Rüdiger Olderog and Krzysztof Rafał Apt. "Fairness in Parallel programs: the Transformational Approach". In: *ACM TOPLAS* 10.3 (July 1988), pp. 420–455 (cit. on pp. 10, 159).

[OG76] Susan S. Owicki and David Gries. "An Axiomatic Proof Technique for Parallel Programs I". In: *Acta Inf.* 6 (1976), pp. 319–340 (cit. on p. 161).

[OMG06] Object Management Group. *CORBA Component Model v4.0*. 2006 (cit. on p. 71).

[Osg] *OSGi Core Release 5*. `http://www.osgi.org`. The OSGi Alliance, 2012 (cit. on pp. 5, 64, 71).

[OSV11] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide, 2nd Edition*. 2nd. USA: Artima Incorporation, 2011 (cit. on p. 1).

[Par76] David Michael Ritchie Park. "Finiteness is Mu-Ineffable". In: *Theor. Comput. Sci.* 3.2 (1976), pp. 173–181 (cit. on p. 156).

[Par81] David Michael Ritchie Park. "Concurrency and Automata on Infinite Sequences". In: *Theor. Comput. Sci.* 1981, pp. 167–183 (cit. on pp. 10, 137, 171).

[Pet62] Carl Adam Petri. “Kommunikation mit Automaten”. ger. PhD thesis. Universität Hamburg, 1962 (cit. on p. 6).

[PHFKW12] Arnd Poetzsch-Heffter, Christoph Feller, Ilham W. Kurnia, and Yannick Welsch. “Model-Based Compatibility Checking of System Modifications”. In: *ISoLA 2012*. LNCS. Springer, 2012, pp. 97–111 (cit. on p. 253).

[PHKF11] Arnd Poetzsch-Heffter, Ilham W. Kurnia, and Christoph Feller. “Verification of Actor Systems Needs Specification Techniques for Strong Causality and Hierarchical Reasoning”. In: *FoVeOOS*. 2011-26. Technische Universität Karlsruhe, 2011, pp. 289–305 (cit. on p. 254).

[Pie02] Benjamin Crawford Pierce. *Types and programming languages*. MIT Press, 2002 (cit. on p. 4).

[Pla13] David Alan Plaisted. “Source-to-Source Translation and Software Engineering”. In: *JSEA Special Issue on Software Dependability* 6.4a (2013), pp. 30–40 (cit. on pp. 5, 29).

[Plo77] Gordon David Plotkin. “LCF Considered as a Programming Language”. In: *Theor. Comput. Sci.* 5.3 (1977), pp. 223–255 (cit. on pp. 7, 41).

[Pnu77] Amir Pnueli. “The Temporal Logic of Programs”. In: *FOCS*. 1977, pp. 46–57 (cit. on pp. 6, 156).

[RD12] Uday S. Reddy and Brian Patrick Dunphy. “An Automata-Theoretic Model of Idealized Algol - (Extended Abstract)”. In: *ICALP (2)*. 2012, pp. 337–350 (cit. on p. 117).

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004 (cit. on pp. 5, 29).

[RK96] Bernhard Rumpe and Cornel Klein. “Automata Describing Object Behavior”. In: *Specification of Behavioral Semantics in Object-Oriented Information Modeling*. Kluwer Academic Publishers, 1996, pp. 265–286 (cit. on p. 117).

[Roe+01] Willem P. de Roever, Frank S. de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Vol. 54. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001 (cit. on p. 2).

[SBM05] Marjan Sirjani, Frank S. de Boer, and Ali Movaghar-Rahimabadi. "Modular Verification of a Component-Based Actor Language". In: *J. UCS* 11.10 (2005), pp. 1695–1717 (cit. on p. 182).

[Sch01] Susanne Schacht. "Formal Reasoning about Actor Programs Using Temporal Logic". In: *Concurrent Object-Oriented Programming and Petri Nets*. 2001, pp. 445–460 (cit. on pp. 156, 181).

[Sch10] Ina Schaefer. "Variability Modelling for Model-Driven Development of Software Product Lines". In: *VAMOS*. 2010, pp. 85–92 (cit. on p. 40).

[SJ11] Marjan Sirjani and Mohammad Mahdi Jaghoori. "Ten Years of Analyzing Actors: Rebeca Experience". In: *Formal Modeling: Actors, Open Systems, Biological Systems*. 2011, pp. 20–56 (cit. on pp. 4, 29, 38, 39, 117, 182).

[SJBA06] Marjan Sirjani, Mohammad Mahdi Jaghoori, Christel Baier, and Farhad Arbab. "Compositional Semantics of an Actor-Based Language Using Constraint Automata". In: *COORDINATION*. 2006, pp. 281–297 (cit. on p. 117).

[SPH10] Jan Schäfer and Arnd Poetzsch-Heffter. "JCoBox: Generalizing Active Objects to Concurrent Components". In: *ECOOP*. LNCS. Springer, 2010, pp. 275–299 (cit. on pp. 38, 40, 72).

[ST02] Scott Fraser Smith and Carolyn L. Talcott. "Specification Diagrams for Actor Systems". In: *Higher-Order and Symbolic Computation* 15.4 (2002), pp. 301–348 (cit. on pp. 155, 182, 183).

[Ste06] Martin Steffen. "Object-Connectivity and Observability for Class-Based, Object-Oriented Languages". 281 pages. Habilitation thesis. Christian-Albrechts-Universität zu Kiel, July 2006 (cit. on p. 69).

[SW01] Davide Sangiorgi and David Walker. *The Pi-Calculus – A Theory of Mobile Processes*. Cambridge University Press, 2001, pp. I–XII, 1–580 (cit. on p. 62).

[Szy98] Clemens Alden Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley-Longman, 1998, pp. I–XVIII, 1–411 (cit. on pp. 4, 63).

[Tal96] Carolyn L. Talcott. "An Actor Rewriting Theory". In: *Electr. Notes Theor. Comput. Sci.* 4 (1996), pp. 361–384 (cit. on p. 60).

[Tal98] Carolyn L. Talcott. “Composable Semantic Models for Actor Theories”. In: *Higher-Order and Symbolic Computation* 11.3 (1998), pp. 281–343 (cit. on pp. 60, 181).

[THK94] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. “An Interaction-Based Language and its Typing System”. In: *PARLE*. 1994, pp. 398–413 (cit. on p. 156).

[TTA04] Prasanna Thati, Carolyn L. Talcott, and Gul Abdulnabi Agha. “Techniques for Executing and Reasoning about Specification Diagrams”. In: *AMAST*. 2004, pp. 521–536 (cit. on p. 181).

[US87] David Ungar and Randall B. Smith. “Self: The Power of Simplicity”. In: *OOPSLA*. 1987, pp. 227–242 (cit. on p. 4).

[VA01] Carlos Arturo Varela and Gul Abdulnabi Agha. “Programming Dynamically Reconfigurable Open Systems with SALSA”. In: *SIGPLAN Notices* 36.12 (2001), pp. 20–34 (cit. on pp. 4, 38).

[Vas92] Vasco Thudichum Vasconcelos. “Trace Semantics for Concurrent Objects”. MA thesis. Keio University, Mar. 1992 (cit. on p. 60).

[VT91] Vasco Thudichum Vasconcelos and Mario Tokoro. “Traces Semantics for Actor Systems”. In: *Object-Based Concurrent Computing*. 1991, pp. 141–162 (cit. on pp. 59, 60).

[WGS87] Jennifer Widom, David Gries, and Fred B. Schneider. “Completeness and Incompleteness of Trace-Based Network Proof Systems”. In: *POPL*. 1987, pp. 27–38 (cit. on p. 159).

[WP14] Yannick Welsch and Arnd Poetzsch-Heffter. “A fully abstract trace-based semantics for reasoning about backward compatibility of class libraries”. In: *Sci. Comput. Program.* 92 (2014), pp. 129–161 (cit. on p. 7).

[YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. “Object-Oriented Concurrent Programming in ABCL/1”. In: *OOPSLA*. 1986, pp. 258–268 (cit. on pp. 4, 38).

[ZWH12] Damien Zufferey, Thomas Wies, and Thomas A. Henzinger. “Ideal Abstractions for Well-Structured Transition Systems”. In: *VMCAI*. 2012, pp. 445–460 (cit. on p. 182).

[Zwi89] Job Zwiers. *Compositionality, Concurrency and Partial Correctness - Proof Theories for Networks of Processes, and Their Relationship*. Vol. 321. LNCS. Springer, 1989 (cit. on pp. 3, 50, 53).

Appendix

APPENDIX A

Glossary

This appendix provides a summary of abbreviations, symbols, predicates, operators, and functions used throughout the report in Tables A.1 to A.3.

Table A.1.: List of abbreviations

Abbreviation	Description
AA	Actor Automata
ABS	Abstract Behavioral Language
ACA	Actor Configuration Automata
CA	Configuration Automata
CCA	Component Configuration Automata
CompA	Component Automata
DIOA	Dynamic I/O Automata
SIOA	Signature I/O Automata

Table A.2.: List of symbols

Symbol	Description
Actors	
$a, b \in \mathbf{A}$	The set of actor names
$\mathbf{D}$	The set of data values (including booleans, integers and strings)
$u \in \mathbf{U}$	The set of futures
$C \in \mathbf{CL}$	The set of class names
$m \in \mathbf{M}$	The set of messages
$e \in \mathbf{E}$	The set of events
$\mathbf{EC}$	The set of event cores
Automaton Model	
$\mathcal{A}$	A signature automaton (of identifier) $\mathcal{A}$
$\mathcal{C}$	A configuration automaton

l	An action (i.e., the transition label)
s	A typical state symbol
$\mathbb{C}$	A configuration
$\mathbb{A}$	A set of SIOA
$\mathbb{S}$	A mapping of an SIOA to its current states
$\mathbb{B}$	A mapping of a set of SIOA to its component status
$\mathbb{E}$	A set of exposed actors
$\mathbb{U}$	A set of futures the environment generated to call methods of actors in the system
α	An execution (fragment)
t	A trace
$v \in \mathbf{V}$	The set of variables

Table A.3.: List of predicates, operators and functions

Functions	**Description**
Data structures	
$s_1 \cdot s_2$ or $s_1 s_2$	Returns the concatenation of sequences s_1 and s_2
$Pref(s)$	Returns the set of prefix of sequence s
s_1 **pr** s_2	Checks if s_1 is a prefix of s_2
$s{\downarrow}X$	Returns the longest subsequence of the sequence s containing elements in X
$s_1 \subseteq s_2$	Checks if sequence s_1 is contained within sequence s_2
$e <_s e'$	Optimistically checks if an element e occurs before e' in sequence s
$\|s\|$	Returns the length of sequence s
$s[n]$	Returns the n-th element of sequence s
e^*	A regular expression that represents a finite sequence of element e's.
$S \cup S'$	Returns the union of sets S and S'
$S - S'$	Returns the largest subset of S not contained in S'
2^S	Returns the power set of set S
$m[x \mapsto y]$	Returns the insertion or update of the key x with value y to map m
$m_1 \cup m_2$	Returns the joined map of maps m_1 and m_2
$diffOn(m_1, m_2, X)$	Checks if maps m_1 and m_2 differ only in the mapping of the keys in X

Actors	
$acq(e)$	Returns all actor names appearing in event e
$ancestors(a)$	Returns the ancestors of actor a
$caller(e)$	Returns the caller of event e, equivalent to $gen(fut(e))$
$creator(a)$	Returns the creator of actor a
$class(a)$	Returns the class of actor a
$class(e)$	Returns the class used in a creation event e
$descendants(a)$	Returns the descendants of actor a
$dirCreate(C)$	Returns the set of classes whose actors can be directly created by an actor of class C
$emitOf(e)$	Returns the emittance event
$eCore(e)$	Returns the event core of event e
$exposed(t, A)$	Returns the set of actors exposed in t to the set of actors A
$fut(e)$	Returns the future of event e
$gen(u)$	Returns the generator of future u
$Gen(a)$	Returns the set of events generated by actor a
$isCreate(e)$	Checks if e is a creation event
$isEmit(e)$	Checks if e is an emittance event
$isCall(e)$	Checks if e is a method call event
$isRet(e)$	Checks if e is a method return event
$isReact(e)$	Checks if e is a reaction event
$msg(e)$	Returns the message of event e
$remCr(t)$	Removes the task identifier k from extended events $k : a \rightarrow a' :$ **new** $C(\overline{p})$ present in trace t
$target(e)$	Returns the target actor of event e
Automaton Framework	
$auts(\mathbb{C})$	Returns the identifiers of the SIOA contained in a configuration
$names(\mathcal{A})$	Returns the names of actors represented by the SIOA
$in(\mathcal{A})(s)$	Input signature of $\mathcal{A}$ in state s
$out(\mathcal{A})(s)$	Output signature of $\mathcal{A}$ in state s
$int(\mathcal{A})(s)$	Internal signature of $\mathcal{A}$ in state s
$\mathcal{A}_1 \parallel \mathcal{A}_2$	Returns the actor-based parallel composition of SIOA $\mathcal{A}_1$, $\mathcal{A}_2$
$execs(\mathcal{A})$	Returns the set of executions of an automaton $\mathcal{A}$
$traces(\mathcal{A})$	Returns the set of traces of an automaton $\mathcal{A}$
$xtraces(\mathcal{A})$	Returns the set of external traces of an automaton $\mathcal{A}$

APPENDIX B

Operational Semantics of αABS

In this appendix, we provide the complete operational semantics of αABS. This semantics is derived from the operational semantics of ABS [Häh+10].

The operational semantics of αABS is defined by reduction rules on *configurations*. For convenience, we shortly repeat the possible forms of a configuration.

$$
\begin{array}{rcll}
K & ::= & a[C,\sigma,l] & \text{actor } a \\
 & | & u[a,\sigma,l,s] & \text{task with future } u \\
 & | & u[a,\sigma,l,v] & \text{completed task with future } u \\
 & | & K \parallel K & \text{composition .}
\end{array}
$$

We abuse the notation for tasks slightly by allowing u to be a fixed value *cons* to represent the constructor that is invoked when an actor is created.

The following two statement constructs is added to describe the reduction rules.

$$s ::= \ldots \mid \texttt{grab} \mid \texttt{release}$$

Figures B.1 to B.3 describe the reduction rules that are applicable to a configuration. The rules in Figure B.1 focus on the sequential composition aspect of the execution of an actor, while the rules in Figures B.2 and B.3 focus on the actor creation and the concurrent aspects. Rule R-RETURN shows how **return**s are converted into values. Executing a **skip** statement has no effect except allowing the next statement to be processed. The rules for conditionals behave in a standard manner depending on the evaluation of the Boolean condition (see rules R-IF1 and R-IF2). Rules R-ASSIGN and R-FASSIGN describe assigning values to local variables and fields, respectively.

The rules in Figures B.2 and B.3 look into how actor creation, message passing and task handling are done. Rules R-NEW and R-FNEW deal with actor creation: a new actor with a fresh name is created. The fields of the new actor are initialized with the parameters of the **new** $C(\overline{val})$ statement. Other fields that are not covered by the class parameters are mapped to default values similar to Java, e.g., an integer variable to the value of `0`, and represented by σ_{init}. The task where the creation statement occurs proceeds to the next statement. For simplicity these

R-RETURN
$$a[C,\sigma',\top] \parallel u[a,\sigma,\top,\texttt{return}\ e;\ \texttt{release}] \rightsquigarrow a[C,\sigma',\bot] \parallel u[a,\sigma,\bot,\mathcal{E}(e)(\sigma'\cup\sigma)]$$

R-SKIP
$$u[a,\sigma,l,\texttt{skip};s] \rightsquigarrow u[a,\sigma,l,s]$$

R-IF1
$$\frac{\mathcal{E}(e)(\sigma'\cup\sigma) = true}{a[C,\sigma',l] \parallel u[a,\sigma,l,\texttt{if}\ e\ s_1\ \texttt{else}\ s_2;s] \rightsquigarrow a[C,\sigma',l] \parallel u[a,\sigma,l,s_1;s]}$$

R-IF2
$$\frac{\mathcal{E}(e)(\sigma'\cup\sigma) = false}{a[C,\sigma',l] \parallel u[a,\sigma,l,\texttt{if}\ e\ s_1\ \texttt{else}\ s_2;s] \rightsquigarrow a[C,\sigma',l] \parallel u[a,\sigma,l,s_2;s]}$$

R-ASSIGN
$$\frac{val = \mathcal{E}(e)(\sigma'\cup\sigma)}{a[C,\sigma',l] \parallel u[a,\sigma,l,x\ :=e;s] \rightsquigarrow a[C,\sigma',l] \parallel u[a,\sigma[x\mapsto val],l,s]}$$

R-FASSIGN
$$\frac{val = \mathcal{E}(e)(\sigma'\cup\sigma)}{a[C,\sigma',l] \parallel u[a,\sigma,l,f\ :=e;s] \rightsquigarrow a[C,\sigma'[f\mapsto val],l] \parallel u[a,\sigma,l,s]}$$

Figure B.1.: Reduction rules of αABS (1)

rules assume that a constructor *constructor*(C) is present, making the constructor to be a special initial task of the created actor. When a constructor is not present, we assign **skip** to be *constructor*(C). Note that the this initial task has the lock, ensuring that other non-constructor tasks can only be executed after the execution of the constructor is finished (via the padded **release** statement). The difference between these two rules lies on whether the name of the new actor is stored in a local variable or in a field.

Rule R-CALL deal with asynchronous method calls. A new task with appropriate initial local variable assignments (σ_{init}) that belongs to the target of the call is composed with the configuration. The new task is only prepared, so it is yet to acquire the lock. The statement the new task processes is based on the method body. The statement that it processes first needs to be transformed by the *repAwait* function. The function *repAwait*(s) replaces all occurrences of **await** g in the statement s to **release**; **grab**; **await** g. This means that executing an **await** statement guarantees that a task relinquishes the lock at least once before continuing. Afterward, the variables representing the parameters, including the self reference construct **this** are subsituted with the appropriate values. The properly initialized statement is now sandwiched between a **grab** and a **release** statements. This is done so the task has a chance to acquire the lock of the actor and once the task finishes executing the method body, it releases the lock and ends with the return result. Rules R-FCALL1 and R-FCALL2 are the same as rule R-CALL except that the futures are stored in the actor's fields. Because the target can be an actor different from the caller (rule R-FCALL1), the target's actor configuration may be needed

R-NEW
$$\frac{a' \text{ fresh} \qquad \overline{val} = \mathcal{E}(\overline{e})(\sigma' \cup \sigma) \qquad \sigma_{a'} = \sigma_{init}[\overline{f} \mapsto \overline{val}] \qquad s_{cons} = constructor(C)}{\begin{array}{c} a[C', \sigma', l] \parallel u[a, \sigma, l, x = \texttt{new}\ C(\overline{e}); s] \rightsquigarrow \\ a'[C, \sigma_{a'}, \top] \parallel a[C', \sigma', l] \parallel u[a, \sigma[x \mapsto a'], l, s] \parallel cons[a, \{\}, \top, s_{cons}; \texttt{release}] \end{array}}$$

R-FNEW
$$\frac{a' \text{ fresh} \qquad \overline{val} = \mathcal{E}(\overline{e})(\sigma' \cup \sigma) \qquad \sigma_{a'} = \sigma_{init}[\overline{f} \mapsto \overline{val}] \qquad s_{cons} = constructor(C)}{\begin{array}{c} a[C', \sigma', l] \parallel u[a, \sigma, l, f := \texttt{new}\ C(\overline{e}); s] \rightsquigarrow \\ a'[C, \sigma_{a'}, \top] \parallel a[C', \sigma'[f \mapsto a'], l] \parallel u[a, \sigma, l, s] \parallel cons[a, \{\}, \top, s_{cons}; \texttt{release}] \end{array}}$$

R-CALL
$$\frac{\begin{array}{ccc} u' \text{ fresh} & s'' = body(m(\overline{x}), C') & a' = \mathcal{E}(e')(\sigma' \cup \sigma) \\ \overline{val} = \mathcal{E}(\overline{e})(\sigma' \cup \sigma) & \sigma_{u'} = \sigma_{init}[\overline{x} \mapsto \overline{val}] & s' = \texttt{grab}; repAwait(s''); \texttt{release} \end{array}}{\begin{array}{c} a'[C', \sigma'', l'] \parallel a[C, \sigma', l] \parallel u[a, \sigma, l, x := e'.m(\overline{e}); s] \rightsquigarrow \\ a'[C', \sigma'', l'] \parallel u'[a', \sigma_{u'}, \bot, s'] \parallel a[C, \sigma', l] \parallel u[a, \sigma[x \mapsto u'], l, s] \end{array}}$$

R-FCALL1
$$\frac{\begin{array}{ccc} u' \text{ fresh} & s'' = body(m(\overline{x}), C') & a' = \mathcal{E}(e')(\sigma' \cup \sigma) \neq a \\ \overline{val} = \mathcal{E}(\overline{e})(\sigma' \cup \sigma) & \sigma_{u'} = \sigma_{init}[\overline{x} \mapsto \overline{val}] & s' = \texttt{grab}; repAwait(s''); \texttt{release} \end{array}}{\begin{array}{c} a'[C', \sigma'', l'] \parallel a[C, \sigma', l] \parallel u[a, \sigma, l, f := e'.m(\overline{e}); s] \rightsquigarrow \\ a'[C', \sigma'', l'] \parallel u'[a', \sigma_{u'}, \bot, s'] \parallel a[C, \sigma'[f \mapsto u'], l] \parallel u[a, \sigma, l, s] \end{array}}$$

R-FCALL2
$$\frac{\begin{array}{ccc} u' \text{ fresh} & s'' = body(m(\overline{x}), C) & \mathcal{E}(e')(\sigma' \cup \sigma) = a \\ \overline{val} = \mathcal{E}(\overline{e})(\sigma' \cup \sigma) & \sigma_{u'} = \sigma_{init}[\overline{x} \mapsto \overline{val}] & s' = \texttt{grab}; repAwait(s''); \texttt{release} \end{array}}{a[C, \sigma', l] \parallel u[a, \sigma, l, f := e'.m(\overline{e}); s] \rightsquigarrow a[C, \sigma'[f \mapsto u'], l] \parallel u[a, \sigma, l, s] \parallel u'[a, \sigma_{u'}, \bot, s']}$$

Figure B.2.: Reduction rules of αABS (2)

to obtain the method body.

Rules R-GRAB and R-RELEASE govern how locks are acquired and released, respectively. When a task needs to acquire a lock and the actor is in the idle state, then the lock can be grabbed. A task that has a lock may release the lock.

The **get** construct is used to obtain the return result of a method call. Rule R-GET reflects this intention by transferring the value of a resolved future to the actor that requests the value. The local variable is updated to store this value. The consequence of rule R-GET is that executing the **get** construct blocks the actor from progressing until the future is resolved. Rule R-FGET is the same as rule R-GET except that the result is stored in a field.

The last construct to be dealt with is the **await** construct. Rules R-AWAIT1, R-

R-GRAB
$$a[C,\sigma,\bot] \parallel u[a,\sigma,\bot,\texttt{grab};s] \rightsquigarrow a[C,\sigma,\top] \parallel u[a,\sigma,\top,s]$$

R-RELEASE
$$a[C,\sigma,\top] \parallel u[a,\sigma,\top,\texttt{release};s] \rightsquigarrow a[C,\sigma,\bot] \parallel u[a,\sigma,\bot,s]$$

R-GET
$$\frac{u' = \mathcal{E}(e)(\sigma' \cup \sigma)}{a[C,\sigma',l] \parallel u[a,\sigma,l,x := e.\texttt{get};s] \parallel u'[a',\sigma',l',val] \rightsquigarrow a[C,\sigma',l] \parallel u[a,\sigma[x \mapsto val],l,s] \parallel u'[a',\sigma',l',val]}$$

R-FGET
$$\frac{u' = \mathcal{E}(e)\sigma' \cup \sigma}{a[C,\sigma',l] \parallel u[a,\sigma,l,f := e.\texttt{get};s] \parallel u'[a',\sigma',l',val] \rightsquigarrow a[C,\sigma'[f \mapsto val],l] \parallel [a,\sigma,l,s] \parallel u'[a',\sigma',l',val]}$$

R-AWAIT1
$$\frac{\mathcal{E}(e)(\sigma' \cup \sigma) = true}{a[C,\sigma',l] \parallel u[a,\sigma,l,\texttt{await}\ e;s] \rightsquigarrow a[C,\sigma',l] \parallel u[a,\sigma,l,s]}$$

R-AWAIT2
$$\frac{\mathcal{E}(e)(\sigma' \cup \sigma) = false}{a[C,\sigma',l] \parallel u[a,\sigma,l,\texttt{await}\ e;s] \rightsquigarrow a[C,\sigma',l] \parallel u[a,\sigma,l,\texttt{release};\texttt{grab};\texttt{await}\ e;s]}$$

R-AWAIT3
$$\frac{u' = \mathcal{E}(v)(\sigma' \cup \sigma)}{u'[a',\sigma'',l',val] \parallel a[C,\sigma',l] \parallel u[a,\sigma,l,\texttt{await}\ v?x;s] \rightsquigarrow u'[a',\sigma'',l',val] \parallel a[C,\sigma',l] \parallel u[a,\sigma[x \mapsto val],l,s]}$$

R-FAWAIT3
$$\frac{u' = \mathcal{E}(v)(\sigma' \cup \sigma)}{u'[a',\sigma'',l',val] \parallel a[C,\sigma',l] \parallel u[a,\sigma,l,\texttt{await}\ v?f;s] \rightsquigarrow u'[a',\sigma'',l',val] \parallel a[C,\sigma'[f \mapsto val],l] \parallel u[a,\sigma,l,s]}$$

R-AWAIT4
$$\frac{u' = \mathcal{E}(v)(\sigma' \cup \sigma)}{u'[a',\sigma'',l',s'] \parallel a[C,\sigma',l] \parallel u[a,\sigma,l,\texttt{await}\ v?v';s] \rightsquigarrow u'[a',\sigma'',l',s'] \parallel a[C,\sigma',l] \parallel u[a,\sigma,l,\texttt{release};\texttt{grab};\texttt{await}\ v?v';s]}$$

Figure B.3.: Reduction rules of αABS (3)

AWAIT2, R-AWAIT3, R-FAWAIT3, and R-AWAIT4 evaluate whether the task can now progress with its computation. Rules R-AWAIT1 and R-AWAIT2 deal with the case where the guard of the **await** statement is a boolean expression e. Similar to the rules for conditionals, if the boolean expression is evaluated to true, then the task can continue its computation. Otherwise, the task must wait until its next chance

to check whether the guard condition is fulfilled. The rest of the rules deal with the guard $v?v'$, where v is evaluated to some future u. When u is resolved, the task fetches the value *val* and stores it in the local variable (R-AWAIT3) or the field variable (R-FAWAIT3). When this condition is not fulfilled, rule R-AWAIT4 is applied whose effect is the same as R-AWAIT2.

The reduction rules above imply that the actors are never destroyed. Neither do tasks. Actors and tasks that are no longer used can be disposed through a garbage collection mechanism, an aspect orthogonal to the functional behavioral characterization we want to obtain.

The initial configuration of the program depends on what part of the program we want to evaluate. The initial configuration may consist only of a designated initial `Main` actor which executes the main method. It can also consist of a single actor entity representing the main interface of a library (such as a `Server` actor in our server example, see Listing 2.1). Below is an example of an initial configuration of a single `Server` actor receiving a request with the query `qu`.

$$s[\texttt{Server}, \{\}, \bot] \parallel \\ u\left[s, \left\{\begin{array}{l} q \mapsto \texttt{qu}, \\ w \mapsto \texttt{null}, \\ u \mapsto \bot, \\ v \mapsto \texttt{Value}_{def} \end{array}\right\}, \bot, \texttt{grab}; \mathit{repAwait}(\mathit{body}(q), \texttt{Server}); \texttt{release}\right]$$

APPENDIX C

Denotational Semantics of αABS

In this chapter, we provide the complete denotational semantics of αABS. They are derived from the denotational semantics of Creol [AD12].

The denotational semantics is represented by a function $[\![]\!]$ which comes in several flavors:

- A mapping from variable declarations, statements or method implementations and states to states and traces
- A mapping from method names, numbers of tasks and states to states and traces
- A mapping from method names and states to states and traces
- A mapping from classes and numbers of instances to traces
- A mapping from actors, classes and programs to traces

The following describes these mappings for a given syntactic constructs. Constructs that are not explained in Section 4.3 are given a short description.
A skip statement neither changes the state of the actor nor produces an event.

$$[\![\texttt{skip}]\!](\sigma) = \{(\sigma, [])\} \tag{C.1}$$

The declaration of a variable v adds a new mapping to the partial function σ

$$[\![T\,x := e]\!](\sigma) = \{(\sigma[x \mapsto \mathcal{E}(e)\sigma], [])\} \tag{C.2}$$

Sequential composition of statements:

$$[\![s_1; s_2]\!](\sigma) = \{(\sigma_2, t_1 \cdot t_2) \mid \exists \sigma_1 : (\sigma_1, t_1) \in [\![s_1]\!](\sigma) \wedge (\sigma_2, t_2) \in [\![s_2]\!](\sigma_1)\} \tag{C.3}$$

The branching **if** statement has two sets of interpretation, following the two possible results of evaluating the Boolean condition.

$$[\![\texttt{if}\ e\ s_1\ \texttt{else}\ s_2]\!](\sigma) = \begin{array}{l} \{(\sigma_1, t) \mid \mathcal{E}(e)\sigma \wedge (\sigma_1, t) \in [\![s_1]\!]\} \cup \\ \{(\sigma_1, t) \mid \neg\mathcal{E}(e)\sigma \wedge (\sigma_1, t) \in [\![s_1]\!]\} \end{array} \tag{C.4}$$

The assignment statement:

$$[\![v := e]\!](\sigma) = \left\{(\sigma', []) \mid \exists val : val = \mathcal{E}(e)\sigma \wedge \sigma' = \sigma[v \mapsto val]\right\} \tag{C.5}$$

The actor creation statement:

$$[\![v := \texttt{new}\ C(\overline{e})]\!](\sigma) = \left\{ (\sigma', t) \;\middle|\; \begin{array}{l} \exists \overline{val}, a : \overline{val} = \mathcal{E}(\overline{e})\sigma \wedge \sigma' = \sigma[v \mapsto a] \wedge \\ \quad created(a) = \mathcal{E}(this)\sigma \wedge class(a) = C \wedge \\ \quad t = \mathcal{E}(me)\sigma : \mathcal{E}(this)\sigma \rightarrow a : \texttt{new}\ C(\overline{val}) \end{array} \right\} \tag{C.6}$$

The method call statement:

$$[\![v := v'.mtd(\overline{e})]\!](\sigma) = \left\{ (\sigma', t) \;\middle|\; \begin{array}{l} \exists a, \overline{val}, i, u : a = \mathcal{E}(v')\sigma \neq null \wedge \\ \quad u = \langle \mathcal{E}(me)\sigma, \langle a, mtd, i \rangle \rangle \wedge \sigma' = \sigma[v \mapsto u] \wedge \\ \quad \overline{val} = \mathcal{E}(\overline{e})\sigma \wedge t = u \rightarrow a : mtd(\overline{val}) \end{array} \right\} \tag{C.7}$$

The **await** statement with future guard:

$$[\![\texttt{await}\ v?v']\!](\sigma) = \left\{ (\sigma', t) \;\middle|\; \begin{array}{l} \exists t', u, k_c, i, a, mtd, val : \\ \quad u = \mathcal{E}(v)\sigma = \langle k_c, \langle a, mtd, i \rangle \rangle \wedge \\ \quad (\mathcal{E}(u)\sigma \neq \mathsf{undef} \implies val = \mathcal{E}(u)\sigma) \wedge \\ \quad t = yield(u, \sigma|_{ca}) \cdot \\ \qquad resume(u, \sigma'|_{ca}) \cdot u \twoheadleftarrow a : mtd \triangleleft val \wedge \\ \quad \sigma'|_k = \sigma|_k[v' \mapsto val] \end{array} \right\} \tag{C.8}$$

The **await** statement with conditional expression guard is a simplified version of the above semantics, because we only need to adjust that the state satisfies the condition.

$$[\![\texttt{await}\ e]\!](\sigma) = \left\{ (\sigma', yield(\sigma|_{ca}) \cdot resume(\sigma|_{ca}) \,\middle|\, \sigma'|_k = \sigma|_k \wedge \mathcal{E}(e)(\sigma') \right\} \tag{C.9}$$

The **get** statement is a variant of the **await** statement with the future guard, where the yield and release events are not needed to simulate the blocking nature of **get**.

$$[\![v := v'.\texttt{get}]\!](\sigma) = \left\{ (\sigma', t) \;\middle|\; \begin{array}{l} \exists t', u, k_c, i, a, mtd, val : \\ \quad u = \mathcal{E}(v')\sigma = \langle k_c, \langle a, mtd, i \rangle \rangle \wedge \\ \quad (\mathcal{E}(u)\sigma \neq \mathsf{undef} \implies val = \mathcal{E}(u)\sigma) \wedge \\ \quad t = u \twoheadleftarrow a : mtd \triangleleft val \wedge \\ \quad \sigma' = \sigma[v \mapsto val, u \mapsto val] \end{array} \right\} \tag{C.10}$$

Based on the semantics of all statements, the semantics of a single execution of a method is as follows:

$$[\![mtd(\overline{x})\{\overline{TF\ y := e_{init}}; s; \textbf{return}\ e\}]\!](\sigma) =$$

$$\left\{ (\sigma_2, t_1 \cdot t \cdot t_2) \;\middle|\; \begin{array}{l} \exists \overline{val}, \sigma_1, u : (\sigma_1, [\,]) \in [\![\overline{TF\ y := e_{init}}]\!](\sigma[\overline{x} \mapsto \overline{val}]) \wedge \\ \quad (\sigma_2, t) \in [\![s]\!](\sigma_1) \wedge u = \langle \mathcal{E}(caller)\sigma, \mathcal{E}(me)\sigma \rangle \wedge \\ \quad t_1 = resume(\mathcal{E}(me)\sigma, \sigma|_{ca}) \cdot u \twoheadrightarrow \mathcal{E}(this)\sigma : mtd(\overline{val}) \wedge \\ \quad t_2 = u \leftarrow \mathcal{E}(this)\sigma : mtd \triangleleft \mathcal{E}(e)\sigma_2 \cdot yield(\mathcal{E}(me)\sigma, \sigma_2|_{ca}) \end{array} \right\} \tag{C.11}$$

The semantics of i tasks of method mtd:

$$[\![mtd, i]\!](\sigma) = \left\{ t \;\middle|\; \begin{array}{l} t{\downarrow}\{\langle \mathcal{E}(this)\sigma, mtd, j\rangle \mid j \in \{1, \ldots, i\}\} = t \wedge \\ \forall j \in \{1, \ldots, i\} : \exists k, a, \overline{val}, \sigma_k, t_k, \sigma'_k : \\ \quad k = \langle a, mtd, j\rangle \wedge \sigma_k = \sigma[me \mapsto k, caller \mapsto a, \overline{f} \mapsto \overline{val}] \wedge \\ \quad (\sigma'_k, t_k) \in [\![mtdDef(class(\mathcal{E}(this)\sigma), mtd)]\!](\sigma_k) \wedge \\ \quad t{\downarrow}\{k\} = t_k \wedge cond_m(t) \end{array} \right\} \tag{C.12}$$

The combined semantics of a method mtd:

$$[\![mtd]\!](\sigma) = \bigcup_{i \in \mathbf{N}} [\![mtd, i]\!](\sigma) \tag{C.13}$$

The semantics of an actor:

$$[\![a]\!] = \left\{ t'' \;\middle|\; \begin{array}{l} \exists t, t' : t{\downarrow}Mtd(C) = t : \\ \quad (\forall mtd \in Mtd(C) : \exists t_{mtd}, \overline{val}, k, a'' : \\ \quad\quad t_{mtd} \in [\![mtd]\!](\sigma) \wedge t{\downarrow}\{mtd\} = t_{mtd}) \wedge \\ \quad t' = yield(\langle a, -, -\rangle, [this \mapsto a, \overline{ca} \mapsto \overline{val}]) \cdot t \wedge cond_a(t') \wedge \\ \quad (\forall a' : a' \neq a \wedge t'{\downarrow}CrEv(a, a') \neq [\,] \implies |t'{\downarrow}CrEv(a, a')| = 1 \wedge \\ \quad\quad \forall e \in CrEv(a, a'), e' : a' \in acq(e') \implies \neg(e' <_{t'} e)) \wedge \\ \quad (\forall e : isReact(e) \wedge e \subseteq t' \wedge emitOf(e) \subseteq t' \implies \\ \quad\quad emitOf(e) <_{t'} e) \wedge \\ \quad t'' \in addInput(k : a'' \to a : \textbf{new}\ C(\overline{val}) \cdot t', a) \wedge a = taskOf(k) \end{array} \right\} \tag{C.14}$$

The semantics of a class C with i instances (actors):

$$[\![C, i]\!] = \left\{ t \;\middle|\; \begin{array}{l} t{\downarrow}\{(C, j) \mid j \in \{1, \ldots, i\}\} = t \wedge \\ \forall j \in \{1, \ldots, i\} : \exists a, t_a \in [\![a]\!] : \\ \quad class(a) = C \wedge ciid(a) = j \wedge t{\downarrow}a = t_a \end{array} \right\} \tag{C.15}$$

The combined semantics of a class C:

$$[\![C]\!] = \bigcup_{i \in \mathbf{N}} [\![C, i]\!] \tag{C.16}$$

The semantics of program P:

$$[\![P]\!] = \left\{ t' \;\middle|\; \begin{array}{l} \texttt{Main} \notin Cls(P) \wedge \\ \forall C \in Cls(P), \exists t_C : t_C \in [\![C]\!] \wedge t{\downarrow}\{C\} = t_C \wedge \\ \exists t, a, u : t{\downarrow}Cls(P) = t \wedge class(a) = \texttt{Main} \wedge gen(u) = a \wedge \\ \quad t' = remCr(u \rightarrow a : main() \cdot t){\downarrow}\mathbf{E} \end{array} \right\} \tag{C.17}$$

About the Author

Name: Ilham W. Kurnia

Education

- 2009–2014: Doctoral student at the Software Technology Group, University of Kaiserslautern, Germany.
- 2006–2008: Master of Science, European Masters in Computational Logic, New University of Lisbon, Portugal and Dresden University of Technology.
  Thesis title: BTSL* Model Checking with Fairness for Reo.
- 2002–2006: Bachelor of Science in Computer Science, University of Indonesia, Indonesia.
  Thesis title: Verification of Horn-Preneel Authentication Protocol Using AVISPA.
- 2001: A Bursary, St. Patrick's College, Wellington, New Zealand.

Publications

A list of my peer-reviewed publications in reverse chronological order:

- Ilham W. Kurnia and Arnd Poetzsch-Heffter. "Verification of Open Concurrent Object Systems". In: *FMCO 2012*. Vol. 7866. LNCS. Springer, 2013, pp. 83–118
- Arnd Poetzsch-Heffter, Christoph Feller, Ilham W. Kurnia, and Yannick Welsch. "Model-Based Compatibility Checking of System Modifications". In: *ISoLA 2012*. LNCS. Springer, 2012, pp. 97–111
- Ilham W. Kurnia and Arnd Poetzsch-Heffter. "A Relational Trace Logic for Simple Hierarchical Actor-Based Component Systems". In: AGERE! '12. Tucson, Arizona, USA: ACM, 2012, pp. 47–58

Other relevant drafts:

- Ilham W. Kurnia and Arnd Poetzsch-Heffter. *A Dynamic Automaton Model for Open Actor-Based Systems*. Tech. rep. University of Kaiserslautern, 2015
- Arnd Poetzsch-Heffter, Ilham W. Kurnia, and Christoph Feller. "Verification of Actor Systems Needs Specification Techniques for Strong Causality and Hierarchical Reasoning". In: *FoVeOOS*. 2011-26. Technische Universität Karlsruhe, 2011, pp. 289–305
- Ilham W. Kurnia, Arnd Poetzsch-Heffter, and Yannick Welsch. "State-based Object Models Are More Abstract Than Trace-based Models: Towards a Unified Specification Framework". In: *Technical Report No. 2010-13*. 2010-13. Karlsruhe, 2010, pp. 268–282