A Game-Theoretic Model for Distributed Programming by Contract

Anders Starcke Henriksen Tom Hvitved Andrzej Filinski

Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen, Denmark
{starcke, hvitved, andrzej}@diku.dk

Abstract: We present an extension of the *programming-by-contract* (PBC) paradigm to a concurrent and distributed environment. Classical PBC is characterized by *absolute conformance* of code to its specification, *assigning blame* in case of failures, and a *hierarchical*, *cooperative* decomposition model – none of which extend naturally to a distributed environment with multiple administrative peers. We therefore propose a more nuanced contract model based on quantifiable *performance* of implementations; *assuming responsibility* for success, and a fundamentally *adversarial* model of system integration, where each component provider is optimizing its behavior locally, with respect to potentially conflicting demands. This model gives rise to a game-theoretic formulation of contract-governed process interactions that supports compositional reasoning about contract conformance.

1 Introduction

We present in this paper a new, foundational approach for extending *programming/design by contract* (PBC) [Mey92] to a concurrent and distributed environment. PBC is a paradigm for specifying and verifying computer programs, typically by means of formal *pre-* and *postconditions* for code fragments. Given a program component c, a precondition A is a predicate over c's inputs (both explicit and implicit) specifying the *requirements* or *assumptions* made by c. If for instance c computes a function on numbers, A could be the requirement that input x satisfies $x \geq 0$. Conversely, a postcondition B is a predicate over both inputs and outputs of c, specifying c's *guarantees* about its outputs, for the given inputs. In the function example, B could specify that the output number r must satisfy $r^2 = x$. A piece of code that satisfies its specification, even in an inefficient or unexpected way (such as returning $r = -\sqrt{x}$), is then said to be *correct*.

The purpose of PBC is to enable modular design and implementation of programs, by establishing detailed specifications for all module interfaces, in such a way that correctness of the whole program (i.e., top-level module) follows from the correctness of all the component modules. In particular, any failure of the whole program to satisfy its specification can ultimately be attributed to a violation of a specific pre- or post-condition. The for-

mer occurs when a *caller* fails to satisfy the input requirements for invoking a submodule, while the latter indicates the failure of the *callee* to satisfy its output guarantee. In both cases, the implementor of the faulty module is the one who is *blamed*, and the module has to be corrected. This paradigm has also been called *the blame game* [WF09], due to the (somewhat degenerate) game-theoretic nature of each implementor's incentive being to avoid getting blamed.

The appeal of PBC derives from its *compositional* nature, meaning that the implementor of a module need not be aware of the entire context in which the module is used. Pre- and postconditions define exactly what the module and its context *expect* from each other, hence when implementing the module nothing else can – or should – be assumed about the environment, and vice versa. For instance, in the numeric example above, it would be wrong of the environment to use the output of c directly as the input for a second invocation of c, even though this latent bug would go undetected if c simply returned $r = \sqrt{x}$.

The purpose of our work is to extend PBC from a classical one-machine setup to a setting where programs run concurrently on (potentially) different machines, owned by different administrative peers, which setting is becoming ever more relevant in the context of *cloud computing* and *software-as-a-service*. Compositionality – as described above – is a crucial feature in PBC, and it becomes even more important for a distributed computation model, in which knowledge of the entire context is not realistic. Existing work on extending pre/postcondition-style specifications to a concurrent setting have been proposed [Hoo94, OG76], but to our knowledge there exist no extensions of the PBC paradigm to a distributed environment.

2 Distributed PBC

In principle, extending PBC to a concurrent, message-passing setting is relatively straightforward. The evident difference from a sequential setting is that pre- and post-conditions must be generalized from one-shot input–output specifications to *communication-protocol* specifications. That is, input requirements now specify what may legitimately be sent *to* a concurrent module or process, while output guarantees capture what must in turn be sent *by* that process. Moreover, both input and output specifications may now in general refer to the entire communication history between the two processes, or some more compact abstraction thereof. Still, no fundamental conceptual changes to the PBC paradigm seem necessary.

On the other hand, a proper account of realistic *distributed* systems does seem to require a complete reassessment of some basic assumptions of PBC. The problematic characteristics here are the notions of *absolute conformance*, *blame assignment*, and the ultimately *cooperative* model of system development.

By absolute conformance we mean that, once a module violates its specification, the "world stops" and no formal guarantees can be given about possible continued execution; the erroneous module must be repaired before the program can be reliably resumed or restarted. In a large-scale distributed environment, however, failures are a fact of life,

and though we do need to assign blame for them, the model must also include a robust specification of the relevant recovery procedures, and how any further failures by both parties are to be accounted for.

Moreover, not all failures are equally serious, and some might even be expected to occur in the course of a typical interaction sequence. We thus prefer a contract conformance model based not on a binary outcome, but on a quantitative measure, making it possible to also uniformly express performance characteristics (such as responsiveness) and relative importance of potentially conflicting specifications. Using an economic metaphor, a module (or rather, the module's implementor) is rewarded by its environment for "good" behavior, and/or penalized for "bad" behavior.

We can recover the usual absolute notion of contract satisfaction as a requirement that a *correct* module's accumulated balance is always non-negative; thus, in particular, such a module will never be the first to break a communication contract. But the key motivation for quantifiable performance contracts is that they support compositional reasoning about module correctness in systems with more than two module producers, as sketched next.

The second problem with traditional PBC is that simple blame assignment is not by itself a proper foundation for composing distributed modules, because the "environment" of a module cannot in general be considered as a monolithic entity that can be blamed as a whole. Consider an example where a company S ("service provider") chooses to implement a web service W_1 using a gateway service W_2 provided by company G ("subcontractor"). If S provides W_1 as a service to some other company C ("client"), then S cannot rely simply on propagating blame to G if W_1 fails as a result of W_2 "failing first". In other words, S is still *responsible* to C for the correct behavior of W_1 – and cannot be excused by W_2 failing. However, G is in turn responsible to S for W_2, which could imply that G has to pay some fine to S each time W_2 fails. And if S is properly organized, this fine will be sufficient to cover the fine that S has to pay to C.

Again, traditional PBC blame propagation can be seen as a degenerate instance of the responsibility model. The difference is that assuming responsibility for satisfying a contract in general involves more than merely being able to deflect all blame for failure. It is an inherently more robust notion, because it requires a module offering a service to explicitly plan for any or all of its subcontractors not fulfilling their nominal ("happy-path") contracts, and ensuring that any penalties imposed by the service's client can ultimately be recovered from the subservices. In particular, a correct module will never make a high-assurance service (i.e., with a high penalty for failure) rely on a low-assurance one.

Finally, by a cooperative decomposition model in traditional PBC we mean that all the module implementors are – to some degree – ultimately working towards a single goal of building a correct system. Thus, if a module specification is ambiguous or incomplete, the implementor will typically still opt to implement the specification in the "intended" way, even if he doesn't explicitly stand to gain anything from it. In particular, he would normally not choose a deliberately suboptimal algorithm for solving a problem, nor intentionally cause minor failures – even if a such failures are nominally allowed by the performance

contract. (For instance, the specification might say that providing a wrong answer to a question is unacceptable, but explicitly declining to answer is a legitimate response – yet simply uniformly refusing to answer would be considered a "bad-faith" implementation.)

In a distributed setting, the implicit assumption of cooperation is not necessarily justified. In the web-service example above it might be *locally optimal* for the subcontractor G to sometimes have W_2 failing (and taking the penalty), if that for instance would make it possible for G to respond to a request from some other (higher-payoff) company O that neither the main contractor S nor the ultimate client C know or care about. Thus, all parties in a distributed system need to recognize that their PBC communication peers may occasionally – or even consistently – defect on contracts, not due to coding errors or legitimate misinterpretation of the specification, but as a deliberate design choice. The adversarial nature also prompts the need for absolute guarantees in contracts (e.g. "must respond after at most 10 seconds") instead of unenforceable promises ("must respond eventually").

In summary of the observations above, we propose an explicitly *game-theoretic* [Men04]. extension of the PBC paradigm. Companies S, G and C above are modeled as *players* (which we also call *entities* or *principals*). Entities are the responsible actors in the distributed environment, and responsibility is codified in *contracts*, which generalize the pre/postcondition-style specifications of PBC. Each contract is a game between two entities, which specifies *what* should be communicated between them, and *when*. This notion respects compositionality, as bilateral contracts only mention the exact part of the context to whom the entity has commitments. The moves of entities in the game are what they communicate to each other in each round. Guarantees are quantified by assigning to each transition of the game state a *payoff*, which can be thought of as the incremental payment to the first player from the second, resulting from the transition. (Since communication games may go on indefinitely, we assign payoffs also to non-terminal states of the games.) Such a payoff may represent either a payment for services properly rendered, or a fine for unsatisfactory performance. The precise game-theoretic formulation of a contract is therefore an *infinite, simultaneous, zero-sum, two-person game*. In general, each player participates in multiple, concurrent games, and aims to maximize his total payoff, rather than to do well in any particular game.

Even though each contract specifies a zero-sum game, and all entities may be assumed to be rational and enter contracts in expectation of a positive payoff, it doesn't necessarily mean that at least one entity has to lose. The reason is that the system is considered *open*, hence the "losing" entity may actually have an overall positive payoff, as a result of some unknown contracts with an unspecified environment, or ultimately with "nature".

A contract describes a *logical* commitment between two entities, but not how communication is enacted *physically*. In order to fulfill its contractual obligations, each entity implements an overall *strategy* for playing all of its communication games. An implementation consists of a set of *processes* for performing actual computation and communication, and a means of *delegation* to other entities. The latter makes it possible to satisfy a logical commitment without doing the actual communication oneself. (In some cases it may in fact not even be possible to be in charge of the communication oneself.)

3 Game-Theoretic Model

We now present the formal model of distributed PBC. Entities play the role of administrative peers, and may enter into communication contracts with each other. Communication is modeled via directed, point-to-point (i.e. non-shared) *links* (usually written $\lambda_1, \lambda_2, \ldots$). The main difference to channel-based communication models (as seen in, e.g., CSP [Hoa78], CCS [Mil89], or the π-calculus [MPW92]) is that message sends are not contingent upon the recipient being willing to receive the message, like in the IOTA model [Bes90]. We can thus reason about contractual interactions by considering only the *traces* of messages actually sent, without complicating the game-theoretic model with an account of *refusals* to receive a message. Higher-level abstractions, such as (potentially shared) channels, can be modeled by explicit contracts about the underlying link-level interactions, involving positive and negative acknowledgments, timeouts, etc. Each link λ has an alphabet $\mathcal{A}_\lambda$ of messages that may be atomically sent over λ, with a distinguished element $\varepsilon \in \mathcal{A}_\lambda$ representing the absence of explicit communication.

Both contracts and processes are formalized as automata, but with somewhat different components, reflecting the difference between game rules and player strategies. For both variants, a *move* m over a finite set of links Λ is a snapshot of what gets communicated on each link of Λ at a given instant in time; that is, m maps each $\lambda \in \Lambda$ to an $m(\lambda) \in \mathcal{A}_\lambda$. The set of all moves over Λ is denoted $\mathcal{M}_\Lambda$ (i.e. $\mathcal{M}_\Lambda = \Pi_{\lambda \in \Lambda} \mathcal{A}_\lambda$).

Definition 3.1 (Contract)**.** *A contract* c *between entities* P *and* A *(player and adversary) is a 5-tuple:*

$$\mathsf{c} = (\Lambda_{\mathsf{PA}}, \Lambda_{\mathsf{AP}}, G, g_0, \rho)$$

where Λ_{PA} *and* Λ_{AP} *are the finite sets of communication links from entity* P *to entity* A *and vice versa,* G *is the (potentially infinite) set of contract states, and* $g_0 \in G$ *is the start state.* $\rho : G \times \mathcal{M}_{\Lambda_{\mathsf{PA}}} \times \mathcal{M}_{\Lambda_{\mathsf{AP}}} \to G \times \mathbb{Q}$ *is the rule function for the game: when the contract is in a state* g*, and the moves on* Λ_{PA} *and* Λ_{AP} *are* m_{P} *and* m_{A}*, let* $(g', k) = \rho(g, m_{\mathsf{P}}, m_{\mathsf{A}})$*; then* g' *is the new contract state, and* k *is the – possibly negative – incremental payoff to* P *from* A*, resulting from the transition.*

A contract, as defined above, evolves in each time unit, based on the chosen moves (m_{P} and m_{A}) of the two players. A move m may consist of simply "doing nothing", in which case $m(\lambda) = \varepsilon$ for all λ. Time units are assumed to be small and fixed; general timing constraints are expressed by means of explicit counters in the game state. (We present contracts in the next section in a condensed, graphical representation, which can be translated to a contract as of Definition 3.1.) Note that there are no "illegal" moves per se: moves in violation of the nominal game rules will typically be assigned a large negative payoff, but the contract remains in a well-defined state, to guide an orderly recovery.

In general, an entity can negotiate a *portfolio* of contracts $\{\mathsf{c}_1, \ldots, \mathsf{c}_n\}$ involving disjoint link sets, and a strategy is achieved by constructing an *implementation*. An implementation consists of zero or more processes $\{\mathsf{a}_1, \ldots, \mathsf{a}_m\}$, together with a possibility of delegating some of the I/O obligations to other entities (that is, routing an incoming link directly to an outgoing one, typically in another contract). A delegation is relevant when it is impossible

(or merely uneconomical) for an entity to fulfill an obligation itself. Figure 1 illustrates the example from the previous section where entity S has negotiated two contracts $\mathsf{c_{SC}}$ and $\mathsf{c_{SG}}$, with entities C and G respectively. The process a is implemented to handle the obligation of receiving on λ_1, while the obligations of sending on λ_2 are delegated to G via λ_4.

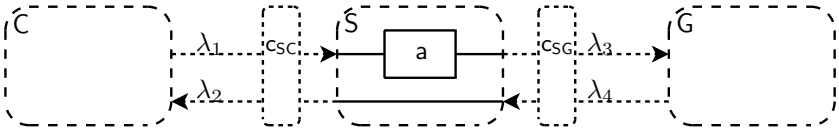

Figure 1: Entities (dashed), contracts/links (dashed) and an implementation (solid).

The system model follows a two-stage approach. In the first stage, the rules of the games (i.e., the contracts) are negotiated. In the second stage, the strategies (i.e., the processes and delegation) are chosen, the network is connected together, and the games begin. There is no changing of network topology or process implementations once gameplay has commenced.

The actual computation and communication in the system is done by *processes*. We do not want to impose a specific computational language for expressing processes, so we also use automata as a suitable abstraction:

Definition 3.2 (Process)**.** *A process (I/O automaton)* a *belonging to an entity* P *is a 6-tuple:*

$$\mathsf{a} = (\Lambda_I, \Lambda_O, S, s_0, \delta_{\mathrm{o}}, \delta_{\mathrm{t}})$$

where Λ_I (input links) and Λ_O (output links) are disjoint and finite, S is the (potentially infinite) set of process states, and $s_0 \in S$ is the start state. $\delta_{\mathrm{o}} : S \to \mathcal{M}_{\Lambda_O}$ and $\delta_{\mathrm{t}} : S \times \mathcal{M}_{\Lambda_I} \to S$ are the output and transition function, respectively: the I/O automaton's output in the current time unit is determined by its internal state, while its next state depends on the current one, and the input received.

As noted above, the output m_o as a reaction to input m_i is only observed in the *next* time unit. Intuitively this means that reaction "takes time", i.e., it it not possible to provide a response instantly (or game-theoretically: a player can only react to the adversary's previous move).

The input and output links of a process come from the links of the contract portfolio of its owning entity, or other processes owned by the entity. The implementation must satisfy that all communication endpoints (input/output links of the contracts/processes) are connected in a sound way (i.e. output to input, no two processes connected to the same link endpoint, etc.) as illustrated in Figure 1.

The processes can be considered as *agents* acting on behalf of the principals (entities). A contract may be handled jointly by multiple agents; and conversely, an agent may be involved in multiple contracts. Note that a process can typically only react on a subset of its owning entity's links: individual agents need not be aware of all information that the

principal may have access to. In particular, they might not know the full game state, or even the game rules. On the other hand, principals nominally do have full knowledge of all their games, but cannot use that information *on-line* to influence their moves, unless the strategy ensures that the relevant data are made available to an agent under the principal's control. In other words, contracts provide an absolute standard against which a module's correctness can be judged (and possibly mathematically proved), but there is no a priori requirement that contract conformance be explicitly monitored at runtime.

4 Example

We now wish to illustrate the concepts of performance, assuming responsibility, and adversarial composition by example. We consider an example with a mobile-phone greeting service, which enables clients to send a predefined greeting card MMS to a specified phone number (for simplicity we fix the greeting card). To send the actual MMS, the service provider has subcontracted with an MMS gateway provider.

The example continues from Figure 1, where S is the service provider (from whose viewpoint we are considering the contracts), C is the client and G is the MMS gateway. As mentioned, the service provider subcontracts with the gateway, and the client has no knowledge of (or interest in knowing) the gateway. Hence the service provider is the one *responsible* to the client for the delivery of the MMS, but the provider has delegated the work to the gateway. Responsibility means that client only needs to be aware of the service provider, not how the provider is organized internally.

The game describing the contract with the client is illustrated in Figure 2 (left). A is the starting state, from which the client can request the MMS greeting. This service has a certain cost, which is modelled as a payoff when the MMS is sent. The service provider guarantees to send the greeting MMS after at most 5 minutes (depicted as a timeout in the figure); otherwise the client gets a discount of twenty percent. If the MMS has not been sent after a total of 15 minutes, the client gets a full refund plus an additional twenty percent of the price. This contract illustrates both performance and absolute guarantees; initially the service provider has a 5 minutes deadline for sending the MMS, however if this deadline is exceeded, then an extended deadline of 10 minutes becomes active, but complying with this deadline is still an indication of "poorer performance".

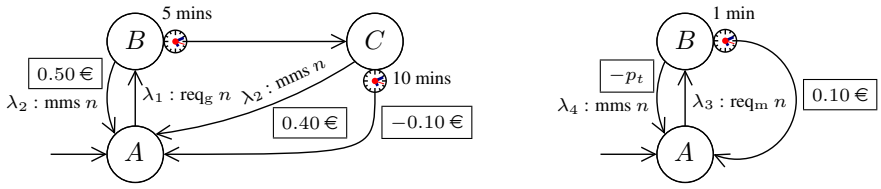

Figure 2: The service provider's contract with the client, c_{SC} (left) and the service provider's contract with the MMS gateway, c_{SG} (right).

The service provider's contract with the MMS gateway is illustrated in Figure 2 (right). The starting state is A, from which the provider can request the sending of an (arbitrary) MMS, at a cost p_t. The subscripted t means that the price depends on the variable t, which refers to the current time (this can be encoded by adding a counter to the contract states, which is incremented in each time unit, cf. Definition 3.1). The value of p_t is defined to be

$$p_t = \begin{cases} 0.15\,€, & 20 \leq hour(t) < 6 \\ 0.30\,€, & \text{otherwise} \end{cases}$$

i.e. if an MMS is requested between 20:00 and 06:00 then the price is 0.15 €, otherwise it is 0.30 € (discount during night hours). If the MMS gateway fails to deliver the MMS within at most one minute, a small penalty is assigned to the MMS gateway.

Now the service provider has to implement a program/strategy (a in Figure 1) for sending MMS greetings for the client. A straightforward implementation is to request sending from the gateway right away upon request from the client, which will produce a profit of either 0.35 € (successful delivery, night hours), 0.20 € (successful delivery, day hours) or 0 € (unsuccessful delivery). But a more clever approach is for the service provider to *game* the client, by deliberately postponing delivery of greetings requested between 19:46 and 20:00; such a strategy will produce a profit of either 0.35 €, 0.25 € or 0 € (as opposed to the straight-forward strategy which will produce either 0.20 € or 0 €). We note that postponing requests received between 19:45 and 19:46 is not safe, since the MMS gateway has a possible delay of one minute, which in the worst case means that a penalty of twenty percent has to be paid to the client and the price of the MMS has to be paid to the gateway.

5 Conformance

We can now formalize what it means for an implementation to be "correct" with respect to a contract portfolio. Since contracts are based on ongoing payoffs, the notion of *contract conformance* is defined as a guarantee of an all-time non-negative accumulated payoff in the games induced by the contract portfolio. As mentioned in Section 2, an implementation consists of a set of processes together with a description of delegation (cf. Figure 1). In the following we assume that delegation is achieved simply by using the same link names in the relevant contracts (e.g. in Figure 1, λ_4 would be renamed to λ_2).

Rather than defining contract conformance with respect to a set of processes, we consider only the case in which an implementation consists of a single process; this simplification is justified by the fact that multiple, parallel processes (which may interact by unobservable communications on internal links) can be described by a single process:

Definition 5.1 (Parallel composition)**.** *Given processes* $\mathsf{a}_1 = (\Lambda_I^1, \Lambda_O^1, S^1, s_0^1, \delta_{\mathrm{o}}^1, \delta_{\mathrm{t}}^1)$ *and* $\mathsf{a}_2 = (\Lambda_I^2.\Lambda_O^2, S^2, s_0^2, \delta_{\mathrm{o}}^2, \delta_{\mathrm{t}}^2)$ *such that* $\Lambda_I^1 \cap \Lambda_I^2 = \Lambda_O^1 \cap \Lambda_O^2 = \emptyset$*, their parallel composition is defined by*

$$\mathsf{a}_1 \parallel \mathsf{a}_2 = (\Lambda_I, \Lambda_O, S^1 \times S^2, \langle s_0^1, s_0^2 \rangle, \delta_{\mathrm{o}}, \delta_{\mathrm{t}})\,,$$

where the links of the composite process are given by:

$$\Lambda_{\text{int}} = (\Lambda_I^1 \cap \Lambda_O^2) \cup (\Lambda_I^2 \cap \Lambda_O^1) \qquad \text{(Internal links)}$$
$$\Lambda_I = (\Lambda_I^1 \cup \Lambda_I^2) \setminus \Lambda_{\text{int}} \qquad \text{(Input links)}$$
$$\Lambda_O = (\Lambda_O^1 \cup \Lambda_O^2) \setminus \Lambda_{\text{int}} \qquad \text{(Output links)}$$

and its output and transition functions:

$$\delta_{\text{o}}(\langle s_1, s_2 \rangle) = (\delta_{\text{o}}^1(s_1) \cup \delta_{\text{o}}^2(s_2))_{|\Lambda_O}$$
$$\delta_{\text{t}}(\langle s_1, s_2 \rangle, m) = \langle \delta_{\text{t}}^1(s_1, (m \cup \delta_{\text{o}}^2(s_2))_{|\Lambda_I^1}), \delta_{\text{t}}^2(s_2, (m \cup \delta_{\text{o}}^1(s_1))_{|\Lambda_I^2}) \rangle$$

(In the above, for a move $m \in \mathcal{M}_\Lambda$, $m_{|\Lambda'}$ denotes the domain restriction of m to Λ'; and moves $m_1 \in \mathcal{M}_{\Lambda_1}$ and $m_2 \in \mathcal{M}_{\Lambda_2}$ over disjoint link sets Λ_1 and Λ_2 are combined as $m_1 \cup m_2 \in \mathcal{M}_{\Lambda_1 \cup \Lambda_2}$.)

Parallel composition can be shown to be commutative and associative with respect to observable behavior; hence when verifying an implementation $\{\mathsf{a}_1, \ldots, \mathsf{a}_m\}$ with respect to a contract portfolio $\{\mathsf{c}_1, \ldots, \mathsf{c}_n\}$, we consider the parallel composition $\mathsf{a}_1 \parallel \cdots \parallel \mathsf{a}_m$ as a single process.

Contract conformance is defined in a coinductive manner – similar to simulations in process calculi [Mil99] – due to the infinite nature of both contracts and processes:

Definition 5.2 (Contract conformance)**.** *Given a process* a*, and a contract portfolio* $\{\mathsf{c}_1, \ldots, \mathsf{c}_n\}$*, where*

$$\mathsf{a} = (\Lambda_I, \Lambda_O, S, s_0, \delta_{\text{o}}, \delta_{\text{t}})$$
$$\mathsf{c}_i = (\Lambda_{\mathsf{PA}_i}, \Lambda_{\mathsf{A}_i\mathsf{P}}, G_i, g_0^i, \rho_i)$$

let $\Lambda = \bigcup_{i=1}^n \Lambda_{\mathsf{A}_i\mathsf{P}}$ be the set of all incoming links from the contracts. A relation $R \subseteq \mathbb{Q} \times S \times G_1 \times \ldots \times G_n$ is said to be a conformance relation for a and $\{\mathsf{c}_1, \ldots, \mathsf{c}_n\}$, if and only if, for all $(k, s, g_1, \ldots, g_n) \in R$ the following holds:

$$\forall m \in \mathcal{M}_{(\Lambda_I \cup \Lambda) \setminus \Lambda_O} .$$
$$(\forall i \in \{1, \ldots, n\} . (g_i', k_i) = \rho_i(g_i, (m \cup \delta_{\text{o}}(s))_{|\Lambda_{\mathsf{PA}_i}}, m_{|\Lambda_{\mathsf{A}_i\mathsf{P}}})) \Rightarrow$$
$$\sum_{i=1}^n k_i \geq k \wedge (k - \sum_{i=1}^n k_i, \delta_{\text{t}}(s, m_{|\Lambda_I}), g_1', \ldots, g_n') \in R$$

The process a is said to conform with contract portfolio $\{\mathsf{c}_1, \ldots, \mathsf{c}_n\}$, written $\models \mathsf{a} : \mathsf{c}_1, \ldots, \mathsf{c}_n$, if there exists a conformance relation R, such that $(0, s_0, g_0^1, \ldots, g_0^n) \in R$.

This definition of $\models \mathsf{a} : \mathsf{c}_1, \ldots, \mathsf{c}_n$ formalizes the guarantee of a consistently non-negative accumulated payoff, when using the strategy a for playing the games induced by the portfolio $\{\mathsf{c}_1, \ldots, \mathsf{c}_n\}$. More generally, whenever a is in state s, contract c_i is in state g_i, and $(k, s, g_1, \ldots, g_n) \in R$, where R is a conformance relation, then the accumulated payoff will remain at least k throughout the remainder of the game. Note also how delegation

is handled by quantifying over all possible inputs m; in the example mentioned earlier, $m(\lambda_2)$ specifies the input from G, and this value is then propagated to the contract with C.

With the definition of conformance it is for instance possible to show that the processes sketched in the previous section (the straightforward strategy and the one which postpones requests between 19:46 and 20:00) both satisfy the portfolio $\{\mathsf{c}_{\mathsf{SG}}, \mathsf{c}_{\mathsf{SC}}\}$.

We mentioned in Section 1 the importance of a compositional extension of PBC to the distributed setting, and in Section 2 we argued how bilateral contracts and assuming responsibility for success achieved this. We now present a result showing that contract conformance is compositional. In other words, the model allows for *internal* compositional reasoning within an entity: to conform with a contract portfolio, the task can be divided between a set of processes. In order to make this intuition formal, we introduce the notion of a *dualized* contract $\bar{\mathsf{c}}$, which is the contract c seen from the viewpoint of the adversary (formally, the links are swapped and the payoff is negated). We can then show:

Theorem 5.3 (Contract conformance is compositional)**.** *Consider two processes* a_1 *and* a_2*, potentially with some links between them. If*

$$\models \mathsf{a}_1 : \mathsf{c}_1, \ldots, \mathsf{c}_n, \mathsf{c}'_1, \ldots, \mathsf{c}'_{n_1}$$
$$\models \mathsf{a}_2 : \overline{\mathsf{c}_1}, \ldots, \overline{\mathsf{c}_n}, \mathsf{c}''_1, \ldots, \mathsf{c}''_{n_2}$$

and the links in the contract portfolio $\{\mathsf{c}'_1, \ldots, \mathsf{c}'_{n_1}, \mathsf{c}''_1, \ldots, \mathsf{c}''_{n_2}\}$ *are disjoint from the internal links of* $\mathsf{a}_1 \parallel \mathsf{a}_2$ *(*Λ_{Int}*, cf. Definition 5.1) then*

$$\models \mathsf{a}_1 \parallel \mathsf{a}_2 : \mathsf{c}'_1, \ldots, \mathsf{c}'_{n_1}, \mathsf{c}''_1, \ldots, \mathsf{c}''_{n_2}$$

This theorem expresses how to fulfill a set of contracts by splitting the obligations between two *contractually compatible* processes. If the set of internal contracts $(\mathsf{c}_1, \ldots, \mathsf{c}_n)$ is empty, then the theorem simply says that two disjoint processes can fulfill a set of contracts by partitioning the set between them. But if the set of internal contracts is non-empty, then these contracts express how the processes can communicate internally to fulfill the external obligations. The duality expresses that they must play opposite roles in the internal contracts.

6 Related and Future Work

In *object-oriented programming*, there have been some investigations of distributed PBC. For example [EC96] is concerned with specifying interfaces for, and behavior of, distributed objects. But it contains no account of multiple administrative entities, which is an integral part of our work.

From a more protocol-oriented perspective, there is relevant work on specifying formal type systems for enforcing correct behavior of concurrent and distributed systems; an example is *session types* [HVK98]. We see such type systems as a pragmatically convenient and concise way of expressing a class of communication contracts that keep track of proper message sequencing, but not full timing or content constraints.

A game-theoretic interpretation of specifications can also be recognized in the theory of *game semantics* [AJ92], where a logical formula or programming-language type determines an abstract, game-like protocol for interacting with values of that type. Our use of games to explicitly model incentive-directed behavior derives much more directly from the original motivation behind game theory.

The conformance relation given in Section 5 gives a semantic characterization of conformance, as indicated by the notation $\models \mathsf{a} : \mathsf{c}_1, \ldots, \mathsf{c}_n$. Current work is concerned with formalizing syntactic proofs of contract conformance, $\vdash \mathsf{a} : \mathsf{c}_1, \ldots, \mathsf{c}_n$, in the tradition of Hoare logic [Hoa69]. A key component in formalizing such proofs is a finitary representation of the mathematical transition functions used in processes and contracts. We intend to look at both a reduced set of contracts, for example those which can be formalized in a certain language, and a more general approach using for example first order logic.

The most important aspects left out in the present work are dynamic contract negotiation and entity/process creation. This means that we only consider a static setup of contracts and peers, without possibility of negotiating contracts in an ongoing fashion; in other words the network topology is fixed. We believe that an extension to a more dynamic setting would not require any fundamental changes to the game-theoretic model, though.

Acknowledgments The authors wish to thank the anonymous SGI reviewers for their numerous comments for improving the presentation.

References

[AJ92] Samson Abramsky and Radha Jagadeesan. Games and Full Completeness for Multiplicative Linear Logic (Extended Abstract). In *Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 291–301, London, UK, 1992. Springer-Verlag.

[Bes90] Azer Bestavros. The Input Output Timed Automaton: A model for real-time parallel computation. In *Proceedings of 1990 ACM Intl. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (Tau '90), Vancouver, Canada, August 1990*, 1990.

[EC96] Chris Exton and Jian Chen. Programming by Contract in a Distributed Object Environment. In *Proceedings of the International Symposium on Future Software Technology (ISFST-96)*, pages 272 – 278. Software Engineers Association (Japan), October 1996.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoo94] Jozef Hooman. Extending Hoare Logic to Real-Time. *Formal Aspects of Computing*, 6:6–801, 1994.

[HVK98] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP'98, volume 1381 of LNCS*, pages 122–138. Springer-Verlag, 1998.

[Men04] Elliot Mendelson. *Introducing Game Theory and its Applications*. Chapman & Hall/CRC, 2004.

[Mey92] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.

[Mil89] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[Mil99] Robing Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Compututation*, 100(1):1–40, 1992.

[OG76] Susan S. Owicki and David Gries. An Axiomatic Proof Technique for Parallel Programs I. *Acta Inf.*, 6:319–340, 1976.

[WF09] Philip Wadler and Robert Bruce Findler. Well-Typed Programs Can't Be Blamed. In Giuseppe Castagna, editor, *18th European Symposium on Programming*, volume 5502 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2009.