Towards Automatic Verification of Coordinated Telecommunication Services

Richard Hull  Grigoris Karvounarakis  Val Tannen

March 8, 2005

Abstract

The e-service paradigm has emerged as a new framework for the design of distributed reactive systems and protocols. Converged networks are expected to allow combinations between telecommunication and web services in the near future. Especially in the telecommunications context, the anticipated abundance of such services further motivates the need for models and languages that allow the design of new services, as well as the coordination of different (existing or new) ones, in order to form richer and more complex “composite” services.

For this reason, we introduce a model that incorporates constructs that - from our experience - are central to coordination of telecommunication services. These constructs could form the basis of a high-level, concrete specification language (a la BPEL), which is natural for programmers, provides the necessary tools for the design of telecommunication services and - up to some extent, as we show later - affords the possibility of automatic verification.

Then, we describe a tangible first step towards automatic verification of this model. Since the expressive power of our model makes most interesting properties undecidable, we resort to approximation techniques: we propose an abstraction mechanism from our service coordination (Minoan) model to an abstract model of counter machines and propose a sufficient condition for proving boundedness of the abstract model. We also show that the abstraction is faithful and as a result if the abstract model is bounded, the concrete specification has to be bounded, too. Finally, specifications that have been shown to be bounded are essentially finite state systems and we expect to be able to use model checking techniques (state space exploration) and tools to verify some of their properties (up to feasibility due to performance limitations posed by the state space explosion problem).

1 Introduction

The web services paradigm (building on standards such as SOAP, WSDL, UDDI, SPEL, OWL-S) promises to bring a revolution in the flexibility of combining distributed processes and services, in order to create more complex services. At the same time, the movement of telecom to IP-based networks and emerging standards such as SIP and 3GPP IMS hold the promise of a broad variety of telecom services and features, which may in principle be combined dynamically with each call completed or message sent. And these worlds are coming together in the form of converged services - in which telecom services incorporate web services (e.g., combining a user’s location with a map) and web services incorporate
telecom services (e.g., during an e-commerce application a message or phone call to a shop owner is launched). In the following we use the term ‘converged’ services to refer to web services and their combinations, to telecom and their combinations, and to combinations of web and telecom services.

A key challenge in converged services is to develop appropriate models of how they interact - models of their purpose, of their message passing, of the activities they perform, and of the data they manipulate. This is clear from the proliferation of standards (e.g., XLANG, WSFL, BPML, BPEL, W3C Choreography WG) and research models (e.g., AZTEC, XL, OWL-S, Conversation model, Roman model, ??? from HP) that embody various aspects of converged services and interaction management. Another issue that is probably going to prove crucial for the success of such models in practice is the ability for automatic analysis and verification of properties of such interactions between services, e.g., being able to reason about the correctness of a specification in terms of some desired behavior. As a result, the need for models that are expressive enough to facilitate application needs but also simple enough to allow the possibility of automatic verification poses an interesting tradeoff/challenge.

In this paper, we propose the Minoan model, that synthesizes key elements of existing standards and research models, which is natural for programmers, provides the necessary tools for the design of telecommunication services and - up to some extent - affords the possibility of automatic verification.

Our model is centered on three principles:

- **Object-based**: The process model permits groupings of activity in *M-instances* of *M-classes*, somewhat analogous to how data and processing in object-oriented systems is grouped around *object instances* of *object classes*. Reminiscent of BPML, and in contrast to BPEL, an arbitrary number of M-instances can be spawned within a single enactment of a Minoan service. With each service we also associate a local data-store.

- **Message-based**: Following the spirit of WSDL and the Conversation model, in the Minoan model the primary means of communication between the M-instances within a service enactment is based on kinds (i.e., classes, types) of messages. These messages also carry data, in the form of parameters. Messages are passed to already running M-instances (in contrast to AZTEC, where all “messages”/events had the effect of spawning a new instance of an “active flowchart”). Note that our model allows asynchronous communication between M-instances, implemented by means of *unbounded* message queues, one for every kind of incoming message coming from a specific sender M-class (possibly corresponding to several M-instances). This way an M-instance can handle “seamlessly” communication/interaction with multiple other services without having to deal with a - potentially - arbitrary order of arrival of messages, due to transmission delays.

- **Finite State Automata based**: Following the spirit of work in the verification community, Process Algebras and the Conversation model, the process model for individual M-classes is based on finite state automata. In particular, our model is that of transducers where transitions are guarded by conditions on data or message receptions and their actions involve sending and receiving messages, spawning M-instances and manipulation of service-instance specific data.
Since this model is too powerful to allow for the automatic verification of most interesting properties, we have to resort to approximation techniques; we propose an abstraction mechanism (i.e., a sound but not complete translation) from our service coordination (Minoan) model to an abstract model of counter machines and propose a sufficient condition for proving boundedness of the abstract model. We also show that the abstraction is faithful and, as a result, if the abstract model is bounded, the concrete specification has to be bounded, too. Finally, we prove that specifications that have been shown to be bounded are essentially finite state systems and we expect to be able to use model checking techniques (state space exploration) and tools to verify some of their properties (up to feasibility due to performance limitations posed by the state space explosion problem).

The rest of the paper is organized as follows: In Section 2 we compare our model to standards and research models for the design of services and their composition. We also recall known decidability results from the verification literature for similar models and justify our decision to use an approximation (abstraction) technique. In Section 3 we present an example of such a composite service and illustrate how distinctive features of our model are useful/necessary/natural for the design of such applications. In Section 4 we give a formal description of our model, which we call the concrete model, as opposed to the abstract model. In section 5.2 we introduce a simplified abstract model and define the translation of a concrete specification to its abstract image. Finally, in Section 6 we define the notion of boundedness, as a form of desirable behavior, and propose a sufficient condition on abstract images that guarantees boundedness of concrete specifications. Moreover, we also give the quantitative aspect of this boundedness, by suggesting a way to compute a (tight) upper bound to the length of any execution.

2 Related Work

Several standards (e.g., XLANG, WSFL, BPML, BPEL, W3C Choreography WG) and research models (e.g., AZTEC, XL, OWL-S, Conversation model, Roman model, ??? from HP) for the description of Web Services and their interactions have been proposed recently. Compared to BPEL, our model supports unbounded spawning, which BPEL does not. On the other hand, BPML does support arbitrary spawning but restricts interprocess communication within a scope, while our model allows any process to send a message to any other. The Conversation model is very close to our model, with the exception of dynamic spawning, which it does not handle. AZTEC is another model whose purpose is the composition of services; however, it is a lot simpler, since it does not allow processes to exchange messages (every message/event results in spawning a new flowchart), although it does support dynamic spawning. Finally, compared to Petri Nets where parallelism is expressed through multiple tokens, our model represents parallelism more explicitly, by allowing an arbitrary number of simultaneous instances; however, this has its drawbacks in allowing automatic verification and one important intuition behind our abstraction function is the need to “replace” multiple instances with numbers that represent them. Indeed, it turns out that our abstract model is quite similar to place/transition Petri Nets.

Formal models of communicating processes has been extensively studied by the verification and programming language (Process Algebras) literature ([1], [5], [7], [8], [13], [14], [15], [10], [9], [12]), in order to study properties of such systems; however, the models that have been proposed are either based on synchronous communication or allow only a limited form of asynchroni-
ity (i.e., through bounded queues). Moreover, none of these models allows the dynamic creation of processes. Our model seems closer to some that have been proposed for Validation of Communication Protocols, which involve dynamic spawning and FIFO queues. Starting from the model of Communicating Finite State Machines [4], it has been proven that even the validation of simple properties as safety/liveness - e.g., by checking for the existence of deadlocks - or checking for unspecified receptions and unbounded communications is undecidable for more than two processes and even one unbounded queue [6] (it is decidable for two processes and exactly one unbounded queue). Positive decidability results for fair reachability analysis ([11]) have been achieved for such protocols, for the restricted case of cyclic protocols\(^1\).

On the other hand, models for which straightforward model checking by state-space exploration is undecidable are common in the verification literature, in the form of infinite state systems ([2])\(^2\). In this context, there are two main approaches that have been proposed, in order to deal with the infiniteness. In symbolic verification, one tries to identify constraints which can be used to represent an infinite set of states; an example is the case of QDDs [3], where regular expressions are used to describe the contents of unbounded queues over all possible executions of a communication protocol. However, in our case such constraints do not seem to arise naturally\(^3\). The alternative direction is that of the approximate abstract-check-refine technique\(^4\): by defining gradual (sound) abstractions on the original model, one hopes to end up with a model for which a desired property can be verified. This approach can, of course, produce false negative answers (this is why it is called an approximation technique); gradual refinement of the abstraction can be used to deal with this problem to some extent: if a negative answer is produced, one can check the same property on a more faithful (i.e., “closer” to the original model) abstraction. The challenge with this technique is to find appropriate abstraction functions for a given model, and our proposal is a first step in this direction (although it does not support any refinement).

3 Motivating Example

Although our model and verification techniques apply for converged services in general, for the sake of simplicity we are going to use as an example an application that only involves telecommunication services; this “advanced teleconference” application is composed by three services: a) a “common” Teleconference Management service, in which all participants call some number and provide a code to connect to the same teleconference, b) a Bridge Management service, which provides an interface to the telecommunication network machinery that is used to setup the actual teleconference, and c) an Autoreconnect service: if a user who is subscribed to this service also participates to a teleconference and they drop from it due to technical reasons (e.g., the user was calling from a cell phone and he drove into a tunnel), then the service is responsible to automatically get them back to the telecon-

---

\(^1\)We must add something about the work of Abdulla/Jonsson and Bouajjani on Lossy FIFO channels, Process Rewrite Systems etc

\(^2\)how does our approach and proof technique compare with well-structured transition systems etc. See abstract of this paper for “monotonicity” as a condition, that looks similar to ours. See also paper by Finkel, I think, on well-structured transition systems. If I remember correct, this notion unifies restrictions for which decidability had been achieved, e.g., lossy channels, but does not prove anything new

\(^3\)although further research could prove the opposite?

\(^4\)find references
ference call (if possible). As is common with telecommunication services, this application will require the coordination of the aforementioned services by a mediator service (in a “hub-and-spoke” topology). This does not imply that our model imposes any such restriction; it could also be used, for instance, to specify composite services that are arranged in a P2P or any other topology, but we are not going to consider them here.

Figure 1 gives a high-level image of the interconnection between the services in a “hub-and-spoke” topology: all interaction is going through the mediator, which is responsible for the coordination of the other services, through the exchange of appropriate messages.

In the following examples observe that some of the parameters in message receptions are preceded by the ‘=’ sign, to denote that only messages whose corresponding part matches the previously assigned value...
The **Telecomm Network** block on the bottom left is not a service (and, thus, there is no M-class specification of its behavior), but we are just using it for the sake of the illustration to show messages that are exchanged between our coordinated service and the underlying telecommunication network.

Figures 2, 3, 4 show the specifications of the corresponding services. Note that a service may include more than one M-class specification (e.g., **Teleconference Manager** and **Add Participant** M-classes both “belong” to the **Teleconference** service). For every teleconference there are as many M-instances of **Mediator.ConferenceRequest** as there are participants who are trying to be added, i.e., the number of M-instances is cannot be bounded. On the other hand, there is only one M-instance of **Teleconference.TeleconferenceManager** that receives a message for every M-instance of **Mediator.ConferenceRequest**; since there is no constraint about how fast these messages are transmitted and how quickly the **Teleconference.TeleconferenceManager** can handle them (which also involves not only removing them from the queue but also some sequence of relevant actions), it is necessary that there is a
queue of unbounded size that holds the messages from the several M-instances of \textit{Mediator.ConferenceRequest}. To complete the outline of an execution of our example of coordinated service, the \textit{TeleconferenceManager} spawns one M-instance of \textit{Teleconference.AddParticipant} for each participant request. These processes communicate with their corresponding \textit{Mediator.ConferenceRequest} M-instance, asking to be connected to the bridge. The mediator M-instances send the appropriate messages to the \textit{BridgeManagement.Main} process and send back acknowledgements to the \textit{AddParticipant} processes, when they receive an acknowledgement from the bridge. The \textit{AddParticipant} processes finish after they receive this acknowledgement. Finally, each M-instance of \textit{Mediator.ConferenceRequest} notifies the autoreconnect service for the addition of the participant to a teleconference, if they are subscribed to the service. For every such participant, an M-instance of \textit{Autoreconnect.Autoreconnect} is spawned; if they drop from the conference, this process will be notified and will take the necessary actions to reconnect them.

Notice that individual services (i.e., Teleconference, Autoreconnect, Bridge) are considered to have been created independently from each other, like “general-purpose” programming modules, and thus cannot communicate directly. The mediator processes are designed specifically to coordinate these other services, based on the behavioral signature they export (in the form of their M-class specifications).

4 Service Coordination Model

We can now define our model formally. In the rest of the paper we are going to refer to this model as the “concrete” model, as opposed to the abstract model, which will be introduced in section 5.2. We define a concrete specification $S_c = \{\Sigma, \mathcal{C}, \eta, \sigma, \tau\}$ where:

- $\Sigma$ is the set of (unique - see note later) message type names.
- $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$ is the set of M-classes
- $\eta : \Sigma \to (\{\rho\} \times DT^*)$ (where $\rho$ is the data type for pids) is a function mapping message types to their signatures (including the recipient pid)
- $\sigma : \Sigma \to \mathcal{C} \cup \{\text{ext}\}$ is the source function, returning the sender class of a message. We also define its inverse: $\sigma^{-1} : \mathcal{C} \to \Sigma^*$ s.t. $\sigma^{-1}(C)$ is the set all messages $m$ for which $\sigma(m) = C$.
- $\tau : \Sigma \to \mathcal{C} \cup \{\text{ext}\}$ is the target function, returning the recipient class of a message. We also define its inverse: $\tau^{-1} : \mathcal{C} \to \Sigma^*$ s.t. $\tau^{-1}(C)$ is the set all messages $m$ for which $\tau(m) = C$.

and $\forall 1 \leq i \leq n, C_i = (\Sigma_{C_i}, T_{C_i}, t^0_{C_i}, F_{C_i}, \Delta_{C_i}, DS_{C_i}, Q_{C_i})$ where:

- $\Sigma_{C_i} = \sigma^{-1}(C_i) \cup \tau^{-1}(C_i)$
- $T_{C_i}$ is the set of states of $C_i$ (and $\forall i, j \in [1 \ldots n], i \neq j \Rightarrow T_{C_i} \cap T_{C_j} = \emptyset$)
- $t^0_{C_i}$ is the initial state of $C_i$
- $F_{C_i}$ is the set of terminating states of $C_i$
\[ Q_{C_i} = \{ q_\sigma | \sigma \in \tau^{-1}(C_i) \} \] with one queue per element of \( \tau^{-1}(C_i) \)

\[ \Delta_{C_i} \subseteq (T_{C_i} \times (\Gamma_c \times A_c) \times T_{C_i}) \]

\( \Gamma_c \) is a finite set of guard expressions \( \gamma = \bigwedge_i \gamma_i > 0 \), where \( \gamma_i \) is either of the form \( C? \sigma(p, \bar{x}) \), \( \sigma \in \Sigma \), \( C \in \mathcal{C} \) or some condition on the data store.

\( A_c \) is a finite set of actions \( \alpha = \alpha_1; \alpha_2; \ldots; \alpha_n \), where \( \alpha_i \) is either of the form \( C! \sigma(p, \bar{x}), \sigma \in \Sigma, C \in \mathcal{C} \) or \( \text{spawn} C, C \in \mathcal{C} \) or some update operation on the data store.

A class specification \( C_i \) is well-typed if:

- For every expression of the form \( C?m(p, \bar{x}) \), \( \tau(m) = C_i \)
- For every expression of the form \( C!m(p, \bar{x}) \), \( \sigma(m) = C_i \)

In the rest of this paper we will only be interested in well-typed specifications.

Note that, wlog we assume that:

- message types are unique (i.e. no two different classes can send or receive the same message). This is “inherent” in the way message types are specified and used: even if the same message type name is used in different classes, message types can be uniquely identified by the triple \( \langle \text{mtype-name, source class, target class} \rangle \). As a result, in the rest of this paper we are going to use \( q_\sigma \) (and later \( mc_\sigma \)) to denote the message queue (counter) associated with message \( \sigma \), ignoring the sender class as a parameter (as used above in \( q(C', \sigma) \)).

- Each transition \( \delta \) contains at most one message reception in its guard (otherwise we can simulate \( \delta \) by adding fresh states and creating a path of consecutive transitions, each of which has a guard with the corresponding reception).

- Each transition \( \delta \) contains at most one message send or process spawn in its action (otherwise we can again introduce fresh states and create a path of consecutive transitions, each of which sends one of the messages or performs one of the spawns).

- Guards are conjunctions of conditions (otherwise we can convert them to DNF and create one transition for each disjunct.

A configuration \( \mu^{\text{inst}} \) of one instance of class \( C_i \), identified by \( \text{pid} \), is a tuple \( (\text{pid} : p, \text{class} : C, \text{state} : t, \text{data} : ds, \text{queues} : v) \) where:

- \( t \) is the state of the machine

- \( ds : DS_{C_i} \to \langle \text{scalar values} \rangle \), represents the contents of the local data store

- \( v : Q_{C_i} \to \eta' \) returns a function \( \eta' \) that given a message type name returns a set of message values (where \( \eta' : \Sigma \to (P \times \text{dom}(DT^*))^* \))
We will use the names \((pid, class, state, data, queues)\) to refer to different “parts” of the tuple.

A configuration \(\mu_c\) of a specification \(S_c\) is a function: \(\mu : P' \rightarrow (P' \times \mathcal{C} \times \mathcal{T} \times \mathcal{D} \times (Q_C \rightarrow \Sigma^*)))\), \(P'\) is finite, \(P' \subseteq P\) (from pids to configurations of the corresponding instances). Abusing notation, we use the shortcut “.” both in \(\mu_c(pid_i).X\) (to refer to “attribute” \(X\) of the tuple \(\mu_c(pid_i)\)) and in \(q.enqueue/dequeue\) (to indicate the corresponding “operation” of the queue).

Let \(V_{S_c}\) denote the (infinite) set of all configurations of \(S_c\). We define a move relation \(\rightarrow_c \subseteq V_{S_c} \times V_{S_c}\) where \(\mu_c \rightarrow_c \mu'_c\) if the following holds:

\[
\exists pid_i \text{ s.t. } \mu_c(pid_i).class = C_x \text{ and } \mu_c(pid_i).state = t \text{ and } \exists \delta \in \Delta_{C_x} \text{ s.t. } \delta = (t, (\gamma \Rightarrow \alpha), t'),
\]

where one of the following two cases hold:

- \(\gamma\) contains a condition of the form \(?^\sigma(p, \vec{x})\) and \(\mu_c(pid_i).v(q_\sigma) \neq \emptyset\) and all conditions on local data are satisfied, and \(\alpha\) contains one of the following:

  - a send operation \(!^\sigma(pid_j, \vec{x}')\), in which case the following update operations take place:

    - State: \(\mu'_c(pid_i).state = t'\)
    - Queues: \(\mu'_c(pid_i).q_\sigma = \mu_c(pid_i).q_\sigma.dequeue(\sigma(pid_i, \vec{\beta}))\)
    - Data: \(\mu'_c(pid_i).data\) reflects (potential) changes in the local store because of other (internal) operations in \(\alpha\)

    (and everything else remains the same between \(\mu_c\) and \(\mu'_c\)).

  - a process spawn operation \(spawn C_x\), in which case the following update operations take place:

    - State: \(\mu'_c(pid_i).state = t'\)
    - Queue: \(\mu'_c(pid_i).q_\sigma = \mu_c(pid_i).q_\sigma.dequeue(\sigma(pid_i, \vec{\beta}))\)
    - Data: \(\mu'_c(pid_i).data\) reflects (potential) changes in the local store because of other (internal) operations in \(\alpha\)
    - New instance: the domain of \(\mu'_c\) contains one new element \(pid_{new}\) s.t.:

      - \(\mu'_c(pid_{new}).pid = newpid()\)
      - \(\mu'_c(pid_{new}).class = C_x\)
      - \(\mu'_c(pid_{new}).state = t'_C\)
      - \(\mu'_c(pid_{new}).data = \emptyset\)
      - \(\forall \sigma \in \Sigma_C, \mu'_c(pid_{new}).q_\sigma = \emptyset\)

    - if \(t' \in F_{C_x}\) the move has the additional effect of finalizing instance \(i\), i.e. \((pid_i) \notin \text{dom}(\mu'_c)\) (the other changes in the configuration are the same with the corresponding case above).

- none of the above, in which case the following update operations take place:

  - State: \(\mu'_c(pid_i).state = t'\)
  - Queue: \(\mu_c(pid_i).q_\sigma = \mu_c(pid_i).q_\sigma.dequeue(\sigma(pid_i, \vec{\beta}))\)
  - Data: \(\mu'_c(pid_i).data\) reflects (potential) changes in the local store of \(C_x\) because of other (internal) operations in \(\alpha\)
does not contain a message reception. Then, for the three cases of actions in \( \alpha \) identified above, the next configuration \( \mu'_c \) is the same as the corresponding case above, without the change involving the dequeueing of message \( \sigma \) (i.e., \( \mu_c(p_{id_i}).q'_\sigma = \mu_c(p_{id_i}).q_\sigma \)).

Note that our model is that of a reactive system rather than that of an automaton - as a result there is no initial state and it is irrelevant to talk about accepting computations etc. However, later in this paper we are also going to further discuss about runs with certain properties.

5 Abstract Model

We present a translation of minoan specifications to \( \pi \)-calculus programs and then an abstraction to rule-based counter programs. Intuitively, the translation involves four aspects:

- We abstract out local data (stores/variables) and all expressions that refer to it\(^6\).
- We translate the resulting model into (a subset of \( ? \)) asynchronous \( \pi \)-calculus.
- We define counters for “atomic” parts (in/out) of the \( \pi \)-calculus programs so that the values of the counters represent the number of “instances” of the programs whose evaluation has brought the corresponding atomic part on top.
- We translate the \( \pi \)-program into a set of rules that update the values of the counters, in order to reflect the transitions (i.e., sequence of applications of reaction rules) in the evaluation of the \( \pi \)-program.

5.1 \( \pi \)-calculus Translation of the Minoan Model

\(^6\)Remark: As it will be more obvious when we discuss about boundedness and verification, when we abstract out the data, there is a useful construct that cannot satisfy our sufficient condition for boundedness (although it could indeed be have bounded behaviour). This construct is loops that are guarded by data variables whose value is updated to reflect the progress in the loop; when we abstract out the data, such loops “look like” unguarded cycles - a definite source of infinite/unbounded behaviour. However, since this functionality is useful we propose an extension to the Minoan model with the introduction of a map operator, to provide a limited form of this functionality, allowing e.g., to send a message to all instances of a class. Adding an appropriate guard condition to those instances could further facilitate sending this message to several (but not necessarily all) instances of that class. This operator could be “implemented” without the use of data variables through spawning: to execute something \( n \) times, spawn \( n \) dummy processes (and maybe a counter with value \( n \)); every time we go through the loop, send a message to one of the dummy processes that makes it move to its terminating state and die (and this way implicitly decreasing the number of dummy processes); set the loop condition to depend on this number of processes (and the other “counter” if added?)

\(^7\)My initial plan was to define a translation to \( \pi \)-calculus before the abstraction and then only talk about the relationship between \( \pi \)/counter programs in the rest of the paper and forget about the minoan model. However, the translation from minoan to \( \pi \) already involves some abstraction, namely losing the order in the queues and - more importantly - disregarding data operations (e.g., conditionals, computations) and queries/updates to local stores (although we could attempt to represent queries/updates as messages on some special channels)
Interestingly, all programs\(^8\) that can be expressed in the Minoan model are definable by a family\(^9\) of asynchronous \(\pi\)-calculus programs, albeit with a small abstraction, i.e., losing the order of exchanged messages in the queues (but this will not matter so much later anyway, when we further abstract to counter programs.

A \(\pi\)-calculus program is defined as a set of processes \(\mathcal{P}\) and a set of channels \(\mathcal{M}\) that the processes are using to communicate.

In order to simplify the presentation of the translation, we consider the following (equally expressive) “variation” of the Minoan model:

- For every \(\delta \in \Delta_{\mathcal{C}}\), s.t. \(\delta = (t_i, (\gamma \Rightarrow \alpha), t'_i), \gamma = C?\sigma(pid, \bar{x})\) and \(\alpha = C'\sigma'(pid', \bar{x}')\) or \(\alpha = \text{spawn } C'(\bar{a}')\) or \(\alpha = \emptyset\)

For the case when there is a transition with \(\gamma = \top\) we are going to assume the existence of a message type \(\text{true()}\) and a process which can send an infinite number of such messages to any other process in order to enable such transitions.

Given this form of M-class concrete specifications, the translation of a concrete specification \(S_0\) to a \(\pi\)-calculus program \(S_\pi\) can be defined as follows (assuming a fixed set of “predefined” channels, i.e., one for every message type name in \(\Sigma\)):

- For every \(C \in \mathcal{C}\), if \(\bar{x}\) is the tuple of parameters of \(C\), we create the program \(C(\bar{x}) := \text{new } p.C_0(p, \bar{x})\), where \(p\) represents the pid of the corresponding M-instance.
- \(\forall t_i \in T_\mathcal{C}\) let \(\Delta_{t_i} \subset \Delta_{\mathcal{C}}\) be the set of transitions whose source is \(t_i\). If \(|\Delta_{t_i}| = n > 1\), then \(C_i(\bar{x}) := C_{i_1}(\bar{x}) + C_{i_2}(\bar{x}) + ... + C_{i_n}(\bar{x})\) \(^{10}\)
- \(\forall \delta_{i_j} \in \Delta_{t_i}\)\(^{11}\)
  - If \(\delta_{i_j} = (t_i, (c_1?m_1(\bar{x}, \bar{y}) \Rightarrow c_2!m_2(\bar{x}, \bar{y}))), t_k\), where \(\bar{x}\) correspond to variables with already assigned values which must match the contents of \(m\) while \(\bar{y}\) are the variables which are bound to some of the contents of \(m_1\), then \(C_{i_j}(\bar{x}) := \text{in } m_1(= \bar{x}, \bar{y}).\text{out } m_2(\bar{x}, \bar{y}).C_k(\bar{x}, \bar{y})\).
  - If \(\delta_{i_j} = (t_i, (c_1?m(\bar{x}, \bar{y}) \Rightarrow \text{spawn } C'(\bar{x}, \bar{y}))), t_k\), where \(\bar{x}\) correspond to variables with already assigned values which must match the contents of \(m\) while \(\bar{y}\) are the variables which are bound to some of the contents of \(m\), then \(C_{i_j}(\bar{x}) := \text{in } m(= \bar{x}, \bar{y}).C'(\bar{x}, \bar{y}).C_k(\bar{x}, \bar{y})\).
  - If \(\delta_{i_j} = (t_i, (c_1?m(\bar{x}, \bar{y}) \Rightarrow \emptyset), t_k\), where \(\bar{x}\) correspond to variables with already assigned values which must match the contents of \(m\) while \(\bar{y}\) are the variables which are bound to some of the contents of \(m\), then \(C_{i_j}(\bar{x}) := \text{in } m(= \bar{x}, \bar{y}).C_k(\bar{x}, \bar{y})\).
  - At the top level, we define a “root” program that contains in parallel:
    * the programs that are considered to be instantiated when the system starts up (e.g., the “root” processes in the teleconference example)

\(^{8}\)after abstracting data operations, too, e.g., conditionals on data values, queries et.c.

\(^{9}\)Should we attempt a more precise characterization of the expressive power of this family - i.e., is it any less expressive than full asynchronous \(\pi\)-calculus

\(^{10}\)where each \(C_{i_j}\) “starts” with an \(\text{in}\)

\(^{11}\)when \(n = 1\) we are going to write \(C_i\) instead of \(C_{i_1}\) for the single program that is created
* for every input message type $x$, as many out $x(y)$ as the number of input messages that we want to simulate

* An arbitrarily large (?) number of out true(), to satisfy the dummy guards that we have introduced.

We are going to illustrate this translation on our example teleconference application that was introduced in section 3. In the following, MCR stands for Mediator.ConferenceRequest (Figure 2(a)), MAR stands for Mediator.AutoreconnectRequest (Figure 2(b)), TR stands for Teleconference.Root (Figure 3(a)), TTM stands for Teleconference.TeleconferenceManager (Figure 3(b)), TAP stands for Teleconference.AddParticipant (Figure 3(c)), BM stands for BridgeManagement (Figure 4(a)), AR stands for Autoreconnect (Figure 4(b))\textsuperscript{12}:

\textsuperscript{12}Note: some values of variables that are sent in messages come from database queries. In the following, I have assumed the existence of a special channel $Q$ that stands for DB queries/updates to assign the values to those variables - more like a placeholder, should be removed/treated more carefully later. Also I am using an assignment ‘:=' as a shortcut for an exchange of messages between a spawner $S$ and a spawned process $P$ in order for the $S$ to receive the process id of the newly created instance of $P$ (i.e., the channel in the new statement of the process $P$) - also to be removed/changed later?
\[ MR() \quad ::= \quad MR_0 \]
\[ MR_0 \quad ::= \quad \text{in setup req}(\text{reqid, devid})(\text{MCR}(\text{reqid, devid})MR_0()) \]
\[ \text{MCR}(\text{reqid, devid}) \quad ::= \quad \text{in true().out conf req} (=\text{"t-root"}, \text{this, reqid, devid})\text{MCR}_1(\text{reqid, devid, this}) \]
\[ \text{MCR}_1(\text{reqid, devid, this}) \quad ::= \quad \text{in new\{\text{conference}\}started}(=\text{this, confid}). \]
\[ \text{MCR}_2(\text{reqid, devid, this, confid}) \quad ::= \quad + \text{in bridged}(=\text{this, confid}). \text{in Q}(\text{bid, = confid})\text{MCR}_3(\text{reqid, devid, this, confid, bid}) \]
\[ \text{MCR}_3(\text{reqid, devid, this, confid, bid}) \quad ::= \quad \text{in add\{part\}conf}(=\text{this, = devid, apid}) \]
\[ \text{MCR}_4(\text{reqid, devid, this, confid, bid, apid}) \quad ::= \quad \text{in leg\{added\}}(=\text{this}). \text{out partic\{added\}(apid, this)}\text{MCR}_5(\text{reqid, devid, this, confid, bid, apid}) \]
\[ \text{MCR}_5(\text{reqid, devid, this, confid, bid, apid}) \quad ::= \quad \text{in true().MAR}(\text{confid, devid})0 \]
\[ \text{MAR}(\text{arid, confid, devid}) \quad ::= \quad \text{new\{this\}MAR}_0(\text{arid, confid, devid, this}) \]
\[ \text{MAR}_0(\text{arid, confid, devid, this}) \quad ::= \quad \text{in autorec\{req\}= this, arid). \text{in Q}(\text{bid, = confid}). \]
\[ \text{MAR}_1(\text{arid, confid, devid, this}) \quad ::= \quad \text{in leg\{added\}}(=\text{this}). \text{out partic\{added\}(arid, devid, confid)}\text{MAR}_0(\text{arid, confid, devid, this}) \]
\[ TR() \quad ::= \quad TR_0() \]
\[ TR_0() \quad ::= \quad \text{in conf\{req\} (=\text{"t-root"}, \text{crid, reqid, devid}). \text{in Q}(\text{confid, = reqid, TR}_1(\text{crid, reqid, devid, confid}) \]
\[ \text{T}_1(\text{crid, reqid, devid, confid}) \quad ::= \quad \text{in Q}(\text{confid, = reqid, in true(), (TTM}(\text{confid, devid})\text{TR}_0()) \]
\[ \text{T}_1(\text{crid, reqid, devid, confid}) \quad ::= \quad \text{in Q}(\text{confid, = reqid, in true(), in Q}(\text{tmpid, = confid, \text{TTM}_0}()) \]
\[ TTM() \quad ::= \quad TTM_0() \]
\[ TTM_0() \quad ::= \quad \text{in true().out conf\{conference\}started}(\text{crid, this})\text{TTM}_1(\text{crid, reqid, devid, this}) \]
\[ \text{TTM}_1(\text{crid, reqid, devid, this}) \quad ::= \quad \text{in true().out apid := TAP}(\text{this, crid, devid})\text{TTM}_2(\text{crid, reqid, devid, this, apid}) \]
\[ \text{TTM}_2(\text{crid, reqid, devid, this, apid}) \quad ::= \quad \text{in conf\{req\}(=\text{this, crid, reqid, devid}). \text{out added\{conf\}(crid, this)}\text{TTM}_1(\text{crid, reqid, devid, this}) \]
\[ \text{TAP}(\text{crid, devid}) \quad ::= \quad \text{new\{this\}TAP}_0(\text{crid, devid, this}) \]
\[ \text{TAP}_0(\text{crid, devid, this}) \quad ::= \quad \text{true().out add\{part\}conf}(\text{crid, devid, this})\text{TAP}_1(\text{crid, devid, this}) \]
\[ \text{TAP}_1(\text{crid, devid, this}) \quad ::= \quad \text{in partic\{added\}}(=\text{this, = crid, 0}) \]
\[ BM() \quad ::= \quad BM_0() \]
\[ BM_0() \quad ::= \quad \text{in setup\{bridge\}(crid, confid). in Q}(\text{bid, = confid, bridge\{set\}(crid, confid, bid)BM}_0()) \]
\[ \text{AR}(\text{confid, devid}) \quad ::= \quad \text{new\{this\}AR}_0(\text{confid, devid, this}) \]
\[ AR_0(\text{confid, devid, this}) \quad ::= \quad \text{in partic\{dropped\}= confid, = devid). in Q}(\text{mar-id, = confid, = devid).} \]
\[ \text{AR}_1(\text{confid, devid, this, mar-id}) \quad ::= \quad \text{in reconnect\{ed\}}(=\text{this, = confid, = devid, AR}_0(\text{confid, devid, this})) \]
\[ P \quad ::= \quad MR(\text{TR(BM)}AR)\text{out\{true\}}() \]

Before describing configurations and moves, we introduce the *projection* $\pi_\alpha$ of a transition in a concrete specification $S_c$ (as defined in Section 4), to its translation in $\pi$-calculus defined above, as follows:

- If $\delta_{ij} = (t_k, (c_i?m_1(x, y) \Rightarrow c_1) \& m_2(x, y))), t_k)$, $\pi_\alpha(\delta_{ij}) = \text{in } m_1(=x, y, \text{out } m_2(x, y))\text{out } m_1(z) \rightarrow \text{out } m_2(x, y)\text{out } C_k(x, y)$.

- If $\delta_{ij} = (t_k, (c_i?m(x, y) \Rightarrow \text{spawn } C'(x, y))), t_k)$, $\pi_\alpha(\delta_{ij}) = \text{in } m(=x, y)\text{out } C'(x, y)\text{C}_k(x, y)\text{out } m(z) \rightarrow C'(x, y)\text{C}_k(x, y)$.

- If $\delta_{ij} = (t_k, (c_i?m(x, y) \Rightarrow \emptyset)), t_k)$, $\pi_\alpha(\delta_{ij}) = \text{in } m(=x, y)\text{C}_k(x, y)\text{out } m(z) \rightarrow C_k(x, y)$.

A configuration $\mu_\pi$ of this machine is a string composed of $\pi$-calculus terms (process names, ins, outs) which correspond to the parallel processes that have come "on top" and are ready for execution at the given time. Let $V_{S_c}$ denote the (infinite) set of all configurations of $S_c$. We define a move relation $\rightarrow_\pi: V_{S_c} \times V_{S_c}$ s.t. $\mu_\pi \rightarrow_\pi \mu'_\pi$ if:
Abusing notation, we also define the projection \( \pi \) of a configuration \( \mu_c \) in the concrete model (as defined in Section 4) to a configuration \( \mu_\pi \) in the \( \pi \)-calculus (as defined in Section 5.1) as follows:

- \( \mu_\pi \) contains a process with a new on top and \( \mu'_\pi \) is the result of extending the scope of the new (i.e., new \( \bar{x}.P|Q \rightarrow \text{new } \bar{x}(P|Q) \)
- \( \mu_\pi \) contains a process name on top and \( \mu'_\pi \) is just the result of replacing the name with its definition (unfolding).
- \( \mu_\pi, \mu'_\pi \) contain the same top level terms in parallel in different order.
- \( \mu_\pi \) contains a term of the form \( A + B \) and \( \mu'_\pi \) is the result of replacing this term with \( B + A \).

Abusing notation, we also define the projection \( \pi_\pi \) of a configuration \( \mu_c \) in the concrete model (as defined in Section 4) to a configuration \( \mu_\pi \) in the \( \pi \)-calculus (as defined in Section 5.1) as follows:

- \( \forall C \in \mathcal{C}, \forall t \in (\mathcal{T}_C - \mathcal{F}_C) \) if \( \{ \{ p \mid p \in \text{dom}(\mu_c) \cap \text{p.class} = C \land p.\text{state} = t_i \} \} \) = \( n \) then \( C_i[C_i] \cdots [C_i] \) is (equivalent up to congruence with) a subsequence of \( \pi_\pi(\mu_c) \).
- \( \forall C \in \mathcal{C}, \) if \( \{ \{ p \mid p \in \text{dom}(\mu_c) \cap \text{p.class} = C \} \} = n, \) then \( \forall q_{\sigma} \in Q_C, \) \( \sum_{i=1}^{n} p.v(q_{\sigma}) \) is a subsequence of \( \pi_\pi(\mu_c) \).

**Lemma 5.1.** (Faithfulness of Translation from Minoan to \( \pi \)-calculus):

\[ \forall \mu_c, \mu'_c \in V_{\mathcal{S}_c} \text{ s.t. } \mu_c \rightarrow^c \mu'_c \Rightarrow \pi(\mu_c) \rightarrow^\pi \mu'_c, \text{ where } \mu'_c \in V_{\mathcal{S}_\pi} \text{ and } \mu'_c \equiv \pi(\mu'_c). \]

**Proof.** By induction on \( n \). Let \( \mu_c = \mu_0, \mu'_c = \mu^{n-1}_c \). We want to show that: \( \mu_0 \rightarrow_c \mu_1 \rightarrow_c \ldots \rightarrow_c \mu_{n-1} \Rightarrow \pi(\mu_0) \rightarrow^\pi \mu_{\pi,1} \rightarrow^\pi \ldots \rightarrow^\pi \mu_{\pi,n-1} \)

where \( \mu_{\pi,1} \equiv \pi(\mu_1), \ldots, \mu_{\pi,n-1} \equiv \pi(\mu_{n-1}) \)

By the induction hypothesis, \( \mu_0 \rightarrow_c \mu_1 \rightarrow_c \ldots \rightarrow_c \mu_{n-2} \Rightarrow \pi(\mu_1) \rightarrow^\pi \mu_{\pi,n-2} \) \hspace{1cm} (1)

where \( \mu_{\pi,n-2} \equiv \pi(\mu_{n-2}) \).
For every individual move between configurations in $V_{S_c}$, by the definitions of the translation to $\pi$-calculus, we observe that for every transition in the concrete specification there exists a corresponding reduction (by firing a redex) in the $\pi$-program. Sometimes the execution will bring on top some process with a new in front, in which case the next execution step of the concrete specification will cannot be directly applied; instead, we have to consider an the equivalent (up to congruence) configuration of the $\pi$-program for which the congruence rule for new has been applied, i.e., extending the scope of the new. Also in another case, the reduction in the $\pi$-calculus program will bring on top a process name and in order to simulate the next step in the concrete specification one first needs to unfold the process definition to which this name corresponds. Still, the name and its unfolding are equivalent up to congruence. (Reordering of parallel and choice do not seem to apply in this proof). (This discussion also justifies the base case of a transition of length 1). As a result:

$$\pi(\mu_{n-2}) \equiv \mu_{\pi,n-2} \rightarrow_\pi \mu_{\pi,n-1} \equiv \pi(\mu_{n-1}) \quad (2)$$

The result follows immediately from (1), (2)

5.2 Counter programs

We define the counter program $S_a$, $(SC \cup MC, \Delta_a)$, where $\Delta_a \subseteq (\Gamma_a \times A_a)$, where:

- $SC, MC$ are sets of state- and message-counters, respectively. We further distinguish between incoming $(MC^{inc})$ and internal $(MC^{int})$ message counters, s.t. $MC = MC^{inc} \cup MC^{int}, MC^{inc} \cap MC^{int} = \emptyset$.
- $\Gamma_a$ is a finite conjunction of guard expressions $\gamma = \bigwedge_{c \in X} c > 0$ (in fact, there is either a single condition or a conjunction of 2) where $X \subseteq SC \cup MC$.
- $A_a$ is a finite set of actions $\alpha = c_1**; c_2**; \ldots c_n**$, where ** stands for either ++ or -- and $c_i \in SC \cup MC$.

At every execution step of this counter program several rules may be enabled (i.e. their guard is satisfied by the values of the counters); only one of those rules is (arbitrarily) chosen for execution and then all guard conditions are re-evaluated in order to determine the rule that will produce the next step.

The translation of a $\pi$-calculus program $S_\pi$ to a counter program $S_a$ can be defined as follows:

- $\forall C \in P \text{ s.t. } C_i(x) ::= \text{in } t = (\tilde{x}, \tilde{y}). C'(\tilde{x}, \tilde{y})'$ there is a (state-)counter denoted as $scc_{C_i} \in SC$. Note that there is one such counter for every syntactic instance of $\text{in } t(...)$ in a different program. For programs of the form $C ::= \text{new } \tilde{x}. C_0$ we are going to consider that $scc$ is the same counter as $scc_0$.

- $\forall t \in M \text{ s.t. } C_i(\tilde{x}) ::= \text{in } s = (\tilde{x}, \tilde{y}). (\text{out } t(\tilde{x}, \tilde{y})|C'(\tilde{x}, \tilde{y}))$ there is a (message-)counter denoted as $mc_{C} \in MC$. Note that if there are multiple syntactic instances of $\text{out } t(\tilde{z})$ for some $\tilde{z}$ in different programs, they all refer to the same message counter.
• ∀Cᵢ, Cⱼ ∈ P s.t. Cᵢ(\(\bar{x}\)) := in t(=\(\bar{x}, \bar{y}\)).C’(\(\bar{x}, \bar{y}\)), Cⱼ(\(\bar{x}\)) := in s(=\(\bar{x}, \bar{y}\)).(out t(\(\bar{x}, \bar{y}\)).C''(\(\bar{x}, \bar{y}\)))¹³

there is a potential reduction from the firing of the redex \(r = Cᵢ(Cⱼ)\); thus, we create one rule in the counter program for each such pair. There are several cases for the form of this rule, depending on the form of C’ (basically two: \((\text{out } u(\bar{z}))C_k\)) or \((D(C_k))\). In all (both) cases the guard of the rule is: \(scCᵢ > 0 \land mcᵢ > 0\). Moreover, in both cases the action includes: \(scCᵢ, mcᵢ\) and:

- if C’ := \(\text{out } u(...))C_k\) (i.e., if the corresponding action in the concrete spec was a send) then the the action also includes the update \(mc_u\). There are also two cases for C_k:
  (a) If C_k := in v(...).D there is also an update \(scC_k\)
  (b) If C_k := C_k₁ + ⋯ + C_kₙ (where C_kᵢ := in v_i(...).D_i) we create n rules with the guard and action as above, adding the counter update \(scC_k\)

- if C’ := D(C_k) (i.e., if the corresponding action in the concrete spec was a spawn) then D := \(\text{new } \bar{x}(D')\) and the action includes the update \(scD\) (i.e., we are “ignoring” the \(\text{new } \bar{x}\) in between). The cases for C_k are the same as (a)\,(b) above.

The translation of our example teleconference π-calculus specification is shown in the following counter program¹⁴:

| \(scMR₀\) > 0 \& \(mc_{cr} > 0\) | \(\Rightarrow \) \(scMR₀, mc_{cr} > 0, scMCR₀, ++, scMR₀, ++\) |
| \(scMCR₀ > 0 \& \(mc_{true} > 0\) | \(\Rightarrow \) \(scMCR₀, mc_{true} > 0, mc_{cr} ++, scMCR₁, ++\) |
| \(scMCR₁ > 0 \& \(mc_{nca} > 0\) | \(\Rightarrow \) \(scMC₁, mc_{nca} > 0, mc_{cr} ++, scMC₁, ++\) |
| \(scMC₁ > 0 \& \(mc_{acte} > 0\) | \(\Rightarrow \) \(scMC₁, mc_{acte} > 0, scMC₁, ++\) |
| \(scMC₂ > 0 \& \(mc_{bc} > 0\) | \(\Rightarrow \) \(scMC₂, mc_{bc} > 0, scMC₂, ++\) |
| \(scMC₃ > 0 \& \(mc_{ap₃c} > 0\) | \(\Rightarrow \) \(scMC₃, mc_{ap₃c} > 0, scMC₃, ++\) |
| \(scMC₄ > 0 \& \(mc_{la} > 0\) | \(\Rightarrow \) \(scMC₄, mc_{la} > 0, mc_{pa} ++, scMC₄, ++\) |
| \(scMC₅ > 0 \& \(mc_{true} > 0\) | \(\Rightarrow \) \(scMC₅, mc_{true} > 0, scMA₀, ++\) |

| \(scMAR₀ > 0 \& \(mc_{ar} > 0\) | \(\Rightarrow \) \(scMAR₀, mc_{ar} > 0, mc_{ap₃c} ++, scMAR₁, ++\) |
| \(scMAR₁ > 0 \& \(mc_{la} > 0\) | \(\Rightarrow \) \(scMAR₁, mc_{la} > 0, mc_{r} ++\) |
| \(scTR₀ > 0 \& \(mc_{cr} > 0\) | \(\Rightarrow \) \(scTR₀, mc_{cr} > 0, scTR₁, ++\) |
| \(scTR₁ > 0 \& \(mc_{true} > 0\) | \(\Rightarrow \) \(scTR₁, mc_{true} > 0, scTTM₀, ++, scTR₀, ++\) |
| \(scTT₀ > 0 \& \(mc_{true} > 0\) | \(\Rightarrow \) \(scTT₀, mc_{true} > 0, scTTM₁, ++\) |
| \(scTT₁ > 0 \& \(mc_{true} > 0\) | \(\Rightarrow \) \(scTT₁, mc_{true} > 0, scTTM₂, ++\) |
| \(scTT₂ > 0 \& \(mc_{cr} > 0\) | \(\Rightarrow \) \(scTT₂, mc_{cr} > 0, scTTM₃, ++\) |
| \(scTT₃ > 0 \& \(mc_{act} > 0\) | \(\Rightarrow \) \(scTT₃, mc_{act} > 0, scTT₄, ++\) |
| \(scTA₀ > 0 \& \(mc_{true} > 0\) | \(\Rightarrow \) \(scTA₀, mc_{true} > 0, scTA₀, ++\) |
| \(scTP₀ > 0 \& \(mc_{pa} > 0\) | \(\Rightarrow \) \(scTP₀, mc_{pa} > 0\) |
| \(scBM₀ > 0 \& \(mc_{ab} > 0\) | \(\Rightarrow \) \(scBM₀, mc_{ab} > 0, scBM₀, ++\) |
| \(scBM₁ > 0 \& \(mc_{al₂b} > 0\) | \(\Rightarrow \) \(scBM₁, mc_{al₂b} > 0, scBM₀, ++\) |
| \(scAR₀ > 0 \& \(mc_{pd} > 0\) | \(\Rightarrow \) \(scAR₀, mc_{pd} > 0, scAR₀, ++\) |
| \(scAR₁ > 0 \& \(mc_{nh} > 0\) | \(\Rightarrow \) \(scAR₁, mc_{nh} > 0, scAR₁, ++\) |

¹³Note that this reduction will only be possible after the leading of \(s(=\bar{x}, \bar{y})\) in \(C_j(\bar{x})\) has taken part in another reduction, leaving \((\text{out } t(\bar{x}, \bar{y})).C''(\bar{x}, \bar{y}))\) on top

¹⁴In this example we are using the first letters of message types for message counter subscripts (e.g. the counter for new_conference_started is mc_nca). I am ignoring in/out Q() statements as well as assignments, since they will probably be removed.
Let the projection \( \pi_a \) of a transition in the \( \pi \)-calculus specification \( S_\pi \) (as defined in Section 5.1), to its abstract image in the abstract model \( S_a \) (as defined in Section 5.2), refer to the counter update operations produced according to the translation presented above.

A configuration \( \mu_a \) of this machine is a function \( \mu_a : SC \cup MC \rightarrow N \) (from a set of state- and message-counters (names) \( (SC \cup MC) \) to a set of integer values \( (N) \)).

Let \( V_a \) denote the (infinite) set of all configurations of \( S_a \). We define a move relation \( \rightarrow_a : V_a \times V_a \) s.t. \( \mu_a \rightarrow_a \mu'_a \) iff \( \exists \delta \in \Delta_a \) s.t. \( \delta = (\gamma \Rightarrow \alpha) \) s.t. \( \gamma \) is satisfied by the values in \( \mu_a \) and:

- \( \alpha \) contains an update \( sc_i-- \) or \( mc_j-- \), then \( \mu'_a(sc_i) = \mu_a(sc_i) - 1 \) or \( \mu'_a(mc_j) = \mu_a(mc_j) - 1 \) respectively.

- \( \alpha \) contains an update \( sc_i++ \) or \( mc_j++ \), then \( \mu'_a(sc_i) = \mu_a(sc_i) + 1 \) or \( \mu'_a(mc_j) = \mu_a(mc_j) + 1 \) respectively.

- Note that \( \alpha \) may contain multiple updates for the same counter (e.g., in the case recursive programs, e.g. \( (P ::= \text{in} \,(x).((\text{out} \,y())|P))|\text{out} \,x() \) results in the rule \( sc_P > 0 \wedge mc_x > 0 \Rightarrow scP --, mcP --, mcP++ , scP++ \). In this case the value of the counter in \( \mu'_a \) is the result of applying all counter updates.

Abusing notation, we also define the projection \( \pi \) of a configuration in the \( \pi \)-calculus model (as defined in Section 5.1) to a configuration in the abstract model (as defined in Section 5.2) as follows:

- \( \forall P \in \mathcal{P} \) if the number of (distinct) processes \( P' \in \mu_\pi \) s.t. \( \mu \subseteq P' \) (i.e., equivalent up to congruence) is \( n \), then \( \mu_\pi(sc_P) = n \), where contains means that either \( P \) itself (as a name) or its unfolding appear in \( \mu_\pi \).

- \( \forall m \in \mathcal{M} \) if \( \{ \text{out } m(...) | \mu_\pi \text{ contains out } m(...) \} = n \), then \( \mu_\pi(mc_m) = n \)

**Lemma 5.2. (Faithfulness of Translation from \( \pi \)-Calculus to Counter Machine):**

\[ \forall \mu_\pi, \mu'_\pi \in V_{S_\pi} \text{ s.t. } \mu_\pi \rightarrow_a^\pi \mu'_\pi \Rightarrow \pi_a(\mu_\pi) \rightarrow_a^\pi \pi_a(\mu'_\pi). \]

**Proof.** By induction on \( n \). Let \( \mu_\pi = \mu^0, \mu'_\pi = \mu^{n-1} \). We want to show that: \( \mu_0 \rightarrow_\pi \mu_1 \rightarrow_\pi \ldots \rightarrow_\pi \mu_{n-1} \Rightarrow \pi_a(\mu_0) \rightarrow_a \pi_a(\mu_1) \rightarrow_a \ldots \rightarrow_a \pi_a(\mu_{n-1}) \)

By the induction hypothesis, \( \mu_0 \rightarrow_c \mu_1 \rightarrow_c \ldots \rightarrow_c \mu_{n-2} \Rightarrow \pi_a(\mu_0) \rightarrow_a \pi_a(\mu_1) \rightarrow_a \ldots \rightarrow_a \pi_a(\mu_{n-2}) \) (1)

For every individual move, by the definitions of \( \Delta_a, \rightarrow_a \) we observe that: \( \forall \mu, \mu' \in S_c \) s.t. \( \mu \rightarrow_c \mu' \) it follows that:

\[ \pi_a(\mu) \rightarrow_a \pi_a(\mu') \] (2)

(The base case, i.e., transition of length 1 also follows from (2))

Letting \( \mu = \mu_{n-2}, \mu' = \mu_{n-1} \) in (2) we get \( \mu_{n-2} \rightarrow_\pi \mu_{n-1} \Rightarrow \pi_a(\mu_{n-2}) \rightarrow_a \pi_a(\mu_{n-1}) \) (3)

The result follows immediately from (1), (3) \qed
6 Boundedness/Quiescence

For the remainder of this paper, fix a concrete specification $S_c$, as defined in Section 4 and let $S_a$ (as defined in Section 4) refer to the abstract image of $S_c$.

We define the input move function $\rightarrow_c \subseteq V_{S_a} \times V_{S_a}$ s.t. (we abuse notation by using $\sigma^{-1}$ on $ext$, which is not a class ...) $\forall \sigma \in \sigma^{-1}(ext), \forall \mu \in V_{S_a}, \mu \rightarrow_c \mu'$ where the only difference between $\mu$, $\mu'$ is that $\exists pid \in P$ s.t. $\mu'(pid).q_\sigma = \mu(pid).q_\sigma$. enqueue($\sigma(ext, \bar{\beta}')$).

For the case of $\pi$-calculus programs, sending input messages does not really correspond to any move in the program; instead, every input message $m$ corresponds to the addition of a parallel process out $m(\bar{x})$ in the “root” $\pi$-calculus program.

Similarly, we define $\rightarrow_a \subseteq V_{S_a} \times V_{S_a}$ s.t. $\forall \sigma \in \sigma^{-1}(ext), \forall \mu \in V_{S_a}, \mu \rightarrow_c \mu'$ where the only difference between $\mu$, $\mu'$ is that $\mu'(mc_\sigma) = \mu(mc_\sigma) + 1$.

We define the extended move function $\rightarrow_{c=} \rightarrow_c \cup \rightarrow_c$. Similarly, $\rightarrow_a = \rightarrow_a \cup \rightarrow_a$.

**Definition:** A configuration $\mu \in V_{S_a}$ is quiescent iff $\exists \mu' \in V_{S_a}$ s.t. $\mu \rightarrow_c \mu'$ - i.e., $\exists C \in C, \exists \delta$ in $\Delta_C$ s.t. $\delta$’s guard is satisfied by $\mu$.

**Definition:** A configuration $\mu \in V_{S_a}$ is quiescent iff $\exists \mu' \in V_{S_a}$ s.t. $\mu \rightarrow_{\pi} \mu'$ - i.e., $\exists$ a pair of parallel processes (or any equivalent rewriting - up to congruence - of any pair of processes) that form a redex.

**Definition:** A configuration $\mu \in V_{S_a}$ is quiescent iff $\exists \mu' \in V_{S_a}$ s.t. $\mu \rightarrow_a \mu'$ - i.e., $\exists$ in $\Delta_a$ whose guard is satisfied by $\mu$.

In the following definitions we use $\ast$ in $V_\ast, S_\ast$ to refer to either of $c$ or $\pi$ or $a$ (the definitions are the same in both cases). Let $V_\ast \subseteq V_\ast$ denote the set of quiescent configurations of $S_\ast$.

We define a run as a sequence $\mu_1, \mu_2, \ldots, \mu_i \in V_{S_a}, i \in \mathbb{N}$ s.t. $\forall i \in \mathbb{N}$ if $i > 0$ then $\mu_i \rightarrow_{\ast} \mu_{i-1}$. A run is finite if this sequence has finite length.

Since input moves are unguarded, $\forall \mu \in V_{S_a}, \exists \mu' \ s.t. \mu \rightarrow_{\ast} \mu'$. As a result, in order for finite runs to be possible, we need to limit the number of input moves. Let a run be finite input iff it contains a finite number of $\rightarrow_{\ast}$-moves. Then, any finite input run finite (or has finite total length) iff it also contains a finite number of $\rightarrow_{\ast}$-moves.

**Definition:** A specification $S_\ast$ is bounded if $\forall \mu \in V_{S_a}$ any finite input run starting from $\mu$ has finite total length.

**Definition:** A specification $S_\ast$ is n-bounded if it is bounded and moreover $\forall \mu \in V_{S_a}$ any finite input run starting from $\mu$ contains at most $n \rightarrow_{\ast}$-moves.

Input moves also have the property of being somewhat independent of the execution of the rest of the rules in that their effect is the same no matter what their interleaving with the rest of the rules is. In particular:

\footnote{Note that $\mu_i$s are not necessarily distinct}

\footnote{i.e. if $\exists \mu' \in V_{S_a}$ s.t. $\mu_n \rightarrow_{\ast} \mu'$, i.e., if $\mu_n$ is quiescent ...}
Lemma 6.1.: Every run starting from \( \mu \) and involving both \( \langle a \rangle \) and \( \langle ! a \rangle \)-moves is equivalent with a run of the same length starting from \( \mu \), where all \( \langle a \rangle \)-moves happen before the first \( \langle ! a \rangle \)-move.

Proof. We want to show that for every pair of moves including a \( \langle a \rangle \)-move we can “switch” their order - if necessary - so that the \( \langle ! a \rangle \)-move happens first. This way, \( \langle a \rangle \)-moves can “float” to the beginning of the run, i.e., before any \( \langle ! a \rangle \)-move has happened.

The only interesting pair of moves appears when a \( \langle ! a \rangle \)-move precedes a \( \langle a \rangle \)-move

\[
\ldots \mu_{i-1} \langle a \rangle \mu_i \langle a \rangle \mu_{i+1} \ldots
\]

We want to show that it is equivalent with:

\[
\ldots \mu_{i-1} \langle a \rangle \mu_i' \langle a \rangle \mu_{i+1} \ldots
\]

We know that input move rules are of the form \( \text{true} \Rightarrow mc++ \), where \( mc \) is a message counter. Observe that any input rule move is satisfied by \( \mu_i \), since they are unguarded, and \( \mu_i' \) only differs from \( \mu_{i-1} \) in that \( \mu_i'(mc) = \mu_{i-1}(mc) + 1 \) \((1)\). Note also that \( \mu_{i+1} \) is produced by \( \mu_{i-1} \) by first applying \( \alpha \) to \( \mu_{i-1} \) and then incrementing \( \mu(mc) \) by 1. \((2)\)

Consider now the rule \( \gamma \Rightarrow \alpha \) that produced the move \( \ldots \mu_{i-1} \langle a \rangle \mu_i \). Its guard, a conjunction of conditions of the form \( c > 0 \), was satisfied, i.e., each of these conjuncts was satisfied by \( \mu_{i-1} \). By \((1)\), they must also be satisfied by \( \mu_i' \) (since \( \mu_i'(mc) > \mu_i(mc) \) and all other counter values remain the same). As a result, \( \gamma \Rightarrow \alpha \) can be applied to \( \mu_i' \), yielding \( \mu_{i+1}' \). However, \( \mu_{i+1}' \) has been produced by applying to \( \mu_{i-1} \) the actions \( mc++ \) and \( \alpha \); since these actions are only additions/subtractions, which are commutative operations, the result does not depend on the order in which they are executed and by \((2)\) above we get \( \mu_{i+1} = \mu_{i+1}' \).

A similar statement can be made for runs involving configurations in \( V_{Sc} \) and the proof is along the same lines.

We define the size of a configuration \( \mu \) as:

\[
\|\mu\| = \sum_{c \in MC \cup SC} \mu(c)
\]

where \( \mu(c) \) is the numeric value of the counter \( c \).

Lemma 6.2.: If there exists a function \( f \) s.t. the length of a run without \( \langle ! a \rangle \)-moves starting from \( \mu \) is bounded by \( f(\|\mu\|) \), then the length of any run starting from \( \mu \), involving \( k \) \( \langle a \rangle \)-moves is bounded by \( k + f(\|\mu\| + k) \).

\(^{17}\)Note that this is also assumed in the translation to \( \pi \)-calculus, where we use a bunch of \( \text{out}'s \) in parallel to simulate input moves, essentially imposing that the input moves happen first. Maybe we need to prove this lemma earlier then for the \( \pi \)-calculus translation. Then, the counter program produced from that does not really have any input moves anymore, since it just starts\(^{18}\) by having some value in the input counters, equal to the number of \( \text{out}'s \) in parallel in the root process.
Proof. By the lemma above, every run is equivalent to a run of the same length, where all \( \rightarrow_s \)-moves happen first. Let \( \mu' \) be the configuration s.t. \( \mu \rightarrow_s k \mu' \). Since every \( \rightarrow_s \)-move has the effect of incrementing one message counter by one, \( \|\mu'\| = \|\mu\| + k \). Then, by our assumption, the length of every run starting from \( \mu' \) is bounded by \( f(\|\mu'\|) = f(\|\mu\| + k) \). Adding the \( k \) input moves yields a total bound of \( k + f(\|\mu\| + k) \).

**Lemma 6.3.** If \( S_a \) is an abstract specification that is the abstract image of concrete specification \( S_c \) and \( S_a \) is bounded, then \( S_c \) is bounded.

Proof. Assume, towards a contradiction, that \( S_a \) is bounded but \( S_c \) is not. By combining the two faithfulness lemmas above, we know that, since \( S_a \) is the abstract image of \( S_c \) (“through” \( S_a \)), for every run \( S_c \) there exists a run of the same length \( S_a \). As a result, if there exists an unbounded run in \( S_c \), then \( S_a \) would also have to be unbounded, which is a contradiction.

### 6.1 Proving boundedness by determining appropriate orderings

We proceed by defining a sufficient condition for determining whether a specification is bounded.

Let \( \preceq \) be a partial ordering of the states of class \( C \). We call \( \preceq \) well-formed if \( \forall \delta = (s, X, t) \in \Delta_C \) either \( s \preceq t \) (in which case we call \( \delta \) a forward transition, relative to \( \preceq \)) or \( t \preceq s \) (backward transition, relative to \( \preceq \)) or \( s = t \) (self-loop transition, relative to \( \preceq \)).

**Lemma 6.4.** \( \forall \) partial ordering \( \preceq \), \( \forall \) non-trivial (>1 edges) cycle in \( \Delta_C \), \( \exists \) (at least) one forward and one backward transition (relative to \( \preceq \)) that belongs to this cycle.

Proof. Obvious

Let \( \ll \) be a partial ordering of \( SC \cup MC \) (counters of \( S_a \)) s.t. \( \forall C \in S \), \( \ll \) is compatible with the partial orderings \( \preceq \) (i.e. the projection of \( \ll \) on the counters of the class \( C \) is equal to \( \preceq \)). Let \( \ll' \) be a total ordering that is compatible with \( \ll \).

Let \( |\mu|_{\ll'} \) denote the “value” of a configuration relative to the total ordering \( \ll' \), constructed by considering the values of the counters in the order of significance\(^{19}\) implied by \( \ll' \). Then, we can compare the values of two configurations by comparing the values of the counters of each in that order (i.e., if \( dom(\mu) = \{c_1, c_2, \ldots, c_n\} \), \( dom(\mu') = \{c_1, c_2, \ldots, c_n\} \) and \( \forall 1 \leq i \leq n, c_i \ll c_{\max} \), if \( \mu'(c_{\max}) < \mu(c_{\max}) \) then \( \mu' \prec_{\ll} \mu \) else ...

Let \( prec_{\ll} \) denote an ordering relation between configurations of \( S_a \), relative to \( \ll \), s.t. \( \forall \) total ordering \( \ll' \) that extends/is compatible with \( \ll \), \( |\mu'|_{\ll'} < |\mu|_{\ll'} \Rightarrow \mu' \prec_{\ll} \mu \).

We call \( \ll \) monotonic if \( \forall \mu, \mu' \in V_a \) if \( \mu \rightarrow_a \mu' \) \( \mu' \prec_{\ll} \mu \).

**Lemma 6.5.** If there exists a monotonic ordering of the state- and message-counters of an abstract specification \( S_a \), then \( S_a \) is bounded.

---

\(^{19}\)Note that, perhaps counter-intuitively, \( c_1 \ll c_2 \) or \( c_1 \ll' c_2 \) means that \( c_1 \) is more significant than \( c_2 \), w.r.t. the lexicographic ordering
We call \( \preceq \) \textit{n-monotonic} if \( \forall \mu, \mu' \text{ s.t. } \mu \rightarrow_A \mu' \text{ it holds that } \mu' \text{ prec}_n \mu \) and \( \forall c \in SC \cup MC, \mu', c \leq \mu, c + n \)

**Observation:** If \( S_a \) is the abstract image of a specification \( S_c \) - in the version of the concrete model without the \textit{map} operator - and \( \ll \) is a \textit{monotonic} ordering for \( S_a \), then \( \ll \) is \textit{1-monotonic}.

An ordering of the state- and message-counters of an abstract specification \( S_a \), as defined above, is well-behaved if it satisfies the following properties\(^{20}\).

1. \( \forall \sigma \in I, \forall c \in SC \cup (MC - I), mc_\sigma \ll c. \)
2. \( \forall C \in \mathcal{C}: \)
   (a) \( \forall \sigma \in \sigma^{-1}(C), \forall sc \in SC_C, sc \ll mc_\sigma. \)
   (b) \( \forall C' \subseteq S \text{ s.t. } C \text{ spawns } C', \forall c \in SC_C \cup MC_C, sc' \in SC_{C'}, c \ll sc'. \)
   (c) \( mc_{true} \) is the least significant of all counters\(^{21}\).
3. \( \forall C \in \mathcal{C}: \)
   (a) Pick a well-formed partial ordering \( \leq_C \) with the property that \( \forall \delta = (t, \gamma \Rightarrow \alpha) \in \Delta_C \) if \( \delta \) is a backward or self-loop transition relative to \( \leq_C \) the abstract image \( \pi(\delta) \) is of the form \( sc_t > 0 \land \gamma \Rightarrow sc_t--; sc_t++; \alpha \) and \( \gamma(\alpha) \) contain a message condition (reception) \( ?m(\ldots) \) then \( mc_m \ll sc_t \)
   (b) if \( t \leq_C t' \) then \( sc_t \ll sc_t' \)

For the case of our example counter program, which indeed always terminates, a total ordering\(^{22}\) of the counters that satisfies the sufficient conditions is the following (where the leftmost counter is the most significant): \( mc_{er} \ll mc_{pd} \ll mc_{nh} \ll mc_{el} \ll sc_{MR_0} \ll sc_{MR_1} \ll sc_{MCR_2} \ll sc_{MCR_1} \ll sc_{MCR_1} \ll sc_{MCR_3} \ll sc_{MCR_3} \ll sc_{MCR_4} \ll mc_{cr} \ll mc_{sb} \ll mc_{pa} \ll sc_{AR_0} \ll sc_{AR_1} \ll mc_{AR} \ll sc_{MAR_0} \ll sc_{MAR_1} \ll mc_{al} \ll mc_{TM_2} \ll sc_{TR_0} \ll sc_{TR_1} \ll mc_{cr} \ll sc_{TM_0} \ll sc_{TM_2} \ll sc_{TM_1} \ll mc_{ncs} \ll mc_{al2e} \ll sc_{TP_0} \ll sc_{TP_1} \ll mc_{ap2e} \ll sc_{BM_0} \ll mc_{bs} \ll mc_{la} \ll mc_{true} \)

Observe that:

- \( sc_{TR_0} \ll sc_{TR_1} \) in order to “break” the cycle
- \( sc_{TM_2} \ll sc_{TM_1} \) in order to “break” the cycle
- \( mc_{ar} \ll sc_{MAR_1} \) (which implies \( sc_{AR} \ll sc_{MAR} \)) in order to “break” the cycle

---

\(^{20}\)The conditions for the \( \pi \)-calculus programs are just an adaptation of the conditions below, according to the translation from concrete to \( \pi \). E.g., \( 2(a) \) becomes: \( \forall C \in calC \) if \( C_i(\ldots) := in t(\ldots) for s(\ldots)(C')(\ldots) \) \( scc_i \ll m_i \) and \( 2(b) \) becomes: \( \forall C, D \in calC \) if \( C_i(\ldots) := in t(\ldots)(D(\ldots)(C')(\ldots)) \) \( scc_i \ll sCD \) Would it be better to present the conditions on \( \pi \)-calculus rather than on minoan specs? The proof below would then need to be adjusted - i.e., the names of the cases would be more like patterns in the \( \pi \)-programs rather than \text{send,recv, spawn} - but apart from that it would remain the same.

\(^{21}\)to reflect the fact that transitions with this guard are essentially unguarded, so it should not be possible to use reception of this message to in order to generate a decreasing step in the counter program

\(^{22}\)not unique, other orderings also exist
We identify the following cases:

- \( mc_{ab} \ll sc_{BM_b} \) (which implies \( sc_{MCR} \ll sc_{BM} \)) in order to “break” the cycle
- \( mc_{c2b} \ll sc_{BM_b} \) (which implies \( sc_{MCR} \ll sc_{BM} \) and \( sc_{MAR} \ll sc_{BM} \)) in order to “break” the cycle
- \( sc_{MR} \ll sc_{MCR} \ll sc_{MAR} \) because of spawning hierarchy
- \( sc_{TR} \ll sc_{TM} \ll sc_{TAP} \) because of spawning hierarchy

Let the projection of \( \ll \) onto \( C \) be equal to (compatible with?) \( \preceq_C \).

The following lemma states that these conditions are sufficient to ensure monotonicity (and, as a result, boundedness of the corresponding specification).

**Lemma 6.6.** Any well-behaved partial ordering of the state- and message-counters of \( S_a \) is monotonic.

**Proof.** Let \( \ll' \) be an arbitrary total ordering that is compatible with \( \ll \) and \( prec_\ll \) is compatible with any such total ordering, as defined above. The proof proceeds by induction on the “structure” a move \( \mu \rightarrow \mu' \), in terms of the kind of the state transition, action and guard. More precisely, we focus on one transition from state \( t \) to \( t' \), potentially guarded by the reception of message \( m \) and potentially producing either a message send (i.e. an increment by 1 of a message-counter) or a process spawn (i.e. an increment by 1 of a state-counter); we use \( \alpha c \) to refer to both cases. As a result, \( |\mu| \) and \( |\mu'| \) only differ at most in these four positions and it is the relative “order of significance” of these counters (according to \( \ll' \)) that determines whether \( \mu' \prec_\ll \mu \). As a result, wlog we will only show the values of the update counters as the value of the configuration in the rest of this proof. Notationally, when we write \( |\mu| = \langle \mu(c_1), \mu(c_2), \ldots, \mu(c_n) \rangle \) we also imply \( c_1 \ll c_2 \ll \ldots \ll c_n \)

We identify the following cases:

- **Case Transition: Forward:** \( \Rightarrow sc_t \ll' sc'_t, sc_t \ll sc'_t, mc \ll mc' \)
  - **Case Guard: Recv from Left:** \( \Rightarrow mc \ll' sc'_t, mc' \]
    - **Case Action: Send:** \( \Rightarrow \alpha c'++ \)
      Observe that, by condition 2.a above, \( sc_t \ll' \alpha c'++, sc'_t \ll' \alpha c'++, \) since \( sc'_t \)
      belongs to the same class as \( sc_t \). \( |\mu| = \langle \mu(mc), \mu(sc_t), \mu(sc'_t), \mu(\alpha c) \rangle, |\mu'| = \langle \mu(mc) - 1, \mu(sc_t) - 1, \mu(sc'_t) + 1, \mu(\alpha c) + 1 \rangle \Rightarrow |\mu'| < |\mu| \Rightarrow \mu' \prec_\ll \mu \)
    - **Case Action: Spawn:** \( \Rightarrow \alpha c'++ \)
      Observe that, by condition 2.b above, \( sc_t \ll' \alpha c'++, sc'_t \ll' \alpha c'++, \) since \( sc'_t \)
      belongs to the same class as \( sc_t \), that spawns the class to which \( \alpha c \) belongs. Then:
      \( |\mu| = \langle \mu(mc), \mu(sc_t), \mu(sc'_t), \mu(\alpha c) \rangle, |\mu'| = \langle \mu(mc) - 1, \mu(sc_t) - 1, \mu(sc'_t) + 1, \mu(\alpha c) + 1 \rangle \Rightarrow |\mu'| < |\mu| \Rightarrow \mu' \prec_\ll \mu \)
  - **Case Guard: Recv from Right:** \( \Rightarrow sc_t \ll' mc, mc' \ll mc' \)
    - **Case Action: Send:** \( \Rightarrow \alpha c'++ \)
      By condition 2.b (repeatedly, for descendant), \( \alpha \ll mc \). Then:
      \( |\mu| = \langle \mu(sc_t), \mu(sc'_t), \mu(\alpha c), \mu(mc) \rangle, |\mu'| = \langle \mu(sc_t) - 1, \mu(sc'_t) + 1, \mu(\alpha c) + 1, \mu(mc) - 1 \rangle \Rightarrow |\mu'| < |\mu| \Rightarrow \mu' \prec_\ll \mu \)

22
- Case Action: Spawn $\Rightarrow \alpha c++$
  In this case, $mc$, $\alpha c$ can appear in any order (or may be incomparable). In all cases (as above) the most significant part ($\mu(sc_t)$) is decreasing $\mu(sc_t) - 1$ $\Rightarrow \mu' \prec \mu$.

- Case Guard: receive from incomparable $\Rightarrow sc_t <> mc, mc--$
  * Case Action: Send $\Rightarrow \alpha c++$
  * Case Action: Spawn $\Rightarrow \alpha c++$
    For both cases of actions, $sc_t \ll' sc'_t \ll' \alpha c++$. If $\mu(sc_t)$ is the most significant part of $\mu \Rightarrow \mu' \prec \mu$, since it decreases as in the previous case. If $\mu(mc)$ is the most significant $\Rightarrow \mu' \prec \mu$, since it also decreases (to $\mu(mc) - 1$). Thus $\mu' \prec \mu$

- Case Guard: true:
  * Case Action: Send $\Rightarrow \alpha c++$
  * Case Action: Spawn $\Rightarrow \alpha c++$
    In both cases $sc_t \ll' sc'_t \ll' \alpha c++$ and $mc$ remains unchanged. As a result, the most significant part that changes is $sc_t$ which decreases $\Rightarrow \mu' \prec \mu$

- Case Transition: Backward $\Rightarrow sc'_t \ll' sc_t, sc_t--, sc'_t++$

- Case Guard: receive from left $\Rightarrow mc \ll' sc_t, mc--$
  * Case Action: Send $\Rightarrow \alpha c++$
  * Case Action: Spawn $\Rightarrow \alpha c++$
    Since $mc$ was sent by a different class than the one to which $sc_t, sc'_t$ belong and $mc \ll' sc_t, mc$ has been sent by an “ancestor” class. Then, by (potentially repeatedly) applying condition 2.a, $mc \ll' sc'_t$. As a result:
    $|\mu| = (\mu(mc, \mu(sc'_t), \mu(sc_t), \mu(\alpha c)), |\mu'| = (\mu(mc) - 1, \mu(sc'_t) + 1, \mu(sc_t) - 1, \mu(\alpha c) + 1) \Rightarrow |\mu'| < |\mu| \Rightarrow \mu' \prec \mu$

- Case Guard: receive from right $\Rightarrow sc_t \ll' mc, mc--$

- Case Guard: receive from incomparable $\Rightarrow sc_t <> mc, mc--$

- Case Guard: true
  These three cases cannot appear, because they contradict conditions 3.a, 3.b.

- Case Transition: self-loop $\Rightarrow sc_t = sc'_t$
  * Case Guard: receive from left $\Rightarrow mc \ll' sc_t, mc--$
    * Case Action: Send $\Rightarrow \alpha c++$
    * Case Action: Spawn $\Rightarrow \alpha c++$
      $mc \ll' sc_t \ll' \alpha c++$ and $sc_t$ remains unchanged:
      $|\mu| = (\mu(mc), \mu(sc_t), \mu(\alpha c)), |\mu'| = (\mu(mc - 1), \mu(sc_t), \mu(\alpha c) + 1) \Rightarrow |\mu'| < |\mu| \Rightarrow \mu' \prec \mu$
  * Case Guard: receive from right $\Rightarrow sc_t \ll' mc, mc--$
  * Case Guard: receive from incomparable $\Rightarrow sc_t <> mc, mc--$
  * Case Guard: true
    These three cases cannot appear, because they contradict conditions 3.a, 3.b.
The following result can be exploited, in order to apply finite state verification (model checking) techniques to verify interesting properties of bounded specifications.

**Lemma 6.7.** If there exists a well-behaved ordering for $S_a$, $S_a$ is finite state.

**Proof.**

As a corollary of the definition of the sufficient condition, one can easily observe the following:

**Corollary 6.8.** If for a specification $S_a$ there is no well-behaved ordering, then $S_a$ is not bounded.

Note that the specification $S_c$, whose $S_a$ is the abstract image, may still be bounded, even if $S_a$ is not.

Finally, the following theorem gives an upper bound on the number of steps that any configuration of a bounded specification with an $n$-monotonic ordering of counters can take, before reaching a quiescent configuration:

**Theorem 6.9.** If there exists an $n$-monotonic ordering of the state- and message-counters of an abstract specification $S_a$, then $S_a$ is $k$-bounded, where $k$ is the number of counters.

**Proof.** Let $S(k, v)$ denote the bound (maximum number of steps) for $k$ counters, each of which has a value $\leq v$. Then, $S(k, v)$ can be defined inductively as follows:

- $S(1, v) = v$, since - in the worst case - in order to get a strictly decreasing sequence, every step will decrease the only counter by 1.
- $S(k, v)$: In the worst case, in a decreasing step in which the value of (most significant) counter $c_k$ decreases, it does so by the minimum amount (i.e., 1), while at the same time the largest possible value (i.e., $n$, because of $n$-monotonicity) is added to every other counter. We call this a “worst-case” step in the strictly decreasing sequence. First, we need to show the following auxiliary lemma:

**Lemma 6.10.** For every strictly decreasing sequence of numbers (formed by sets of counters $C$, ordered according to an $n$-monotonic ordering, s.t. $|C| = k$ and $c_k$ is the most significant counter), a sequence in which steps that decrease the value of $c_k$ are replaced by no more than $v$ “worst-case” steps that are taken before any other step, is at least as long as the original sequence.

**Proof.** (Sketch) Consider a sequence that starts with the number $|\mu|$ and ends with $|\mu_2|$. Let $|\mu_1|, |\mu_2|$ be two consecutive numbers in the sequence s.t. $\mu_1(c_k) > \mu_2(c_k)$. We “replace” $|\mu_2|$ with the number produced by a “worst-case” decreasing step starting from $|\mu_1|$, namely:

$$
\mu'_2 = \mu_1(c_k) - 1 \quad \mu_1(c_{k-1}) + n \quad \ldots \quad \mu_1(c_0) + n
$$
Clearly, $\mu_1 > \mu'_2 \geq \mu_2$, because of n-monotonicity. Suppose there are $m$ such steps, then $m \leq \mu(c_k) \leq v$, since every step decreases $c_k$ by at least 1 and no other step can increase it later (since it is part of a decreasing sequence and $c_k$ is the most significant counter, i.e., there is no way to produce a smaller number if its value is increased). Since each one of the added “worst-case” steps decreases $c_k$ by 1 (i.e., by the minimum amount), we can be sure that these steps will not decrease the value of $c_k$ below zero (otherwise the original sequence would also cause the value of $c_k$ to drop below zero, which is absurd). Suppose that these $m$ decreasing steps were all taken at the beginning of the sequence, then at the end we are left with $|\mu'_m|$ s.t. $\forall i \in [1 \ldots k-1], \mu'_m(c_i) = \mu(c_i) + m \cdot n$.

Consider now the leftmost step from $\mu_a$ to $\mu_b$ of the original sequence, in which $\mu_a(c_k) = \mu_b(c_k)$, and suppose some (e.g., $m' \leq m$) of the old steps that were decreasing $c_k$ used to appear in the sequence before this step. Let $\mu'_a$ be the corresponding number (before this step) in the new sequence. Observe that:

$$\forall i, \mu_a(c_i) = \mu(c_i) + \sum_{l=1}^{m'} v_l \leq \mu(c_i) + m \cdot v = \mu'_a(c_i)$$

As a result, any decrease to the value of any counter that used to take place at this step can still be applied, without making the value of the counter negative, and moreover $|\mu'_b| \geq |\mu_b|$.

Next we pick the leftmost step of the part of the sequence starting from $|\mu_b|$, e.g., from $|\mu_c|$ to $|\mu_d|$, s.t. $\mu_c(c_k) = \mu_d(c_k)$). Similarly:

$$\forall i, \mu_c(c_i) = \mu(c_i) + \sum_{l=1}^{m''} v_l + ab_i \leq \mu(c_i) + m \cdot v + ab_i = \mu'_c(c_i)$$

where $ab_i$ is the change to the value of counter $c_i$ that was produced by the step from $|\mu_a|$ to $|\mu_b|$ (which is equal to the change produced by the step from $|\mu'_a|$ to $|\mu'_b|$ in the new sequence).

Repeating this process for every step until we get to the end of the sequence, we conclude that every step in the old sequence corresponds to a step in the new sequence, i.e., the new sequence is at least as long as the old one.

Using this lemma, the longest possible strictly decreasing sequence can be produced by, first, decreasing the most significant counter value to 0 and increasing all other counters by $\mu(c_k) \cdot n \leq v \cdot n$. Then, the same lemma can be applied again for this decreasing sequence on the set of $k-1$ counters (since the value of $c_k$ is 0 and cannot change for the rest of the sequence); as a result of the first application, each counter has a value $\leq v + v \cdot n = v \cdot (n+1)$: Consequently, $S(k, v) \leq v + S(k-1, v \cdot (n+1))$

Iterating this inductive definition we get:
\[
S(k, v) \leq v + S(k - 1, v \cdot (n + 1))
\]
\[
S(k - 1, v \cdot (n + 1)) \leq v \cdot (n + 1) + S(k - 1, v \cdot (n + 1)^2)
\]
\[
\vdots
\]
\[
+ S(1, v \cdot (n + 1)^{k-1}) \leq v \cdot (n + 1)^{k-1}
\]
\[
S(k, v) \leq v + v \cdot (n + 1) + \cdots + v \cdot (n + 1)^{k-1} = v \cdot \frac{(n + 1)^{k-1} - 1}{n}
\]

Since \( v \leq \|\mu\| \), this gives a bound of \( \|\mu\| \cdot \frac{(n + 1)^{k-1} - 1}{n} \). \(\square\)

References


A Appendix

A.1 Some Notation - Symbols’ Index

In the rest of this paper the meaning of several symbols is as follows:

1. $C$: Minoan Class (and its corresponding FSA)
2. $\mathcal{C}$: set of all M-classes
3. $\text{ext}$: special symbol that denotes an "external" message source, from which input messages are sent
4. $\Sigma$: set of message type names
5. $I$: set of input message type names
6. $DT$: set of data types
7. $P$: set of class instance ids (pids)
8. $\Sigma' = (\Sigma, \eta)$
9. $T_C$: set of states of an M-class
10. $F_C$: set of terminating states of an M-class
11. $Q_C$: set of message queues of a class
12. • $\Gamma_C$: set of guards in concrete model (conjunction of one msg recv and conditions on local vars)
   • $\Gamma_a$: set of guards in model $a$ (-data stores and conditions on data, -pids and parameters from message receptions, -message receptions /+conditions on state counters (of the form: $sc > 0$), +conditions on message counters (of the form $mc > 0$))
13. • $\mathcal{A}_C$: set of actions in concrete model (sequence of one msg send or one process spawn and operations on local vars)
   • $\mathcal{A}_a$: set of actions in model $a$ (-data stores, pids and expressions on them, -process spawns, -msg sends/+state counter update operations, +message counter update operations)
14. $DS_C$: set of variable names
   • $DS_C$: set of local variables (local store) of class $C$
15. $\mathcal{P}$: set of all $\pi$-calculus programs
16. $\mathcal{M}$: set of all $\pi$-calculus channels
17. $SC$: set of state-counters for this specification
   • $SC_C$: set of state-counters of M-Class $C$
18. $MC$: set of message-counters for this specification
   
   • $MC_C$: set of message-counters of M-Class $C$

19. $newpid()$: returns a fresh, unique process (instance) identifier