Translation-Based Compositional Reasoning for Software Systems ⋆

Fei Xie[1], James C. Browne[1], and Robert P. Kurshan[2]

[1] Dept. of Computer Sciences, Univ. of Texas at Austin, Austin, TX 78712, USA
Email: {feixie, browne}@cs.utexas.edu Fax: +1 (512) 471-8885
[2] Cadence Design Systems, 35 Spring St., New Providence, NJ 07974, USA
Email: rkurshan@cadence.com Fax: +1 (908) 898-1435

Abstract. Software systems are often model checked by translating them into a directly model-checkable formalism. Any serious software system requires application of compositional reasoning to overcome the computational complexity of model checking. This paper presents Translation-Based Compositional Reasoning (TBCR), an approach to application of compositional reasoning in the context of model checking software systems through model translation. In this approach, given a translation from a software semantics to a directly model-checkable formal semantics, a compositional reasoning rule is established in the software semantics and mapped to an equivalent rule in the formal semantics based on the translation. The correctness proof of the composition reasoning rule in the software semantics is established based on this mapping and the correctness proof of the equivalent rule in the formal semantics. The compositional reasoning rule in the software semantics is implemented and applied based on the translation from the software semantics to the formal semantics and reusing the implementation of the equivalent rule in the formal semantics. TBCR has been realized for a commonly used software semantics, the Asynchronous Interleaving Message-passing semantics. TBCR is illustrated by two applications of this realization.

Keywords. Translation-based compositional reasoning, model checking, compositional reasoning, model translation

1 Introduction and Overview

Model checking [1–3] has major potential for improving reliability of software systems. Model checking is often applied to software systems by translating them into a model-checkable formalism to avoid the difficulty and labor of developing special-purpose model checkers.

On account of the intrinsic computational complexity of model checking, we need to support compositional reasoning [4–9] where model checking a property on a system is accomplished by decomposing the system into components, checking component properties locally on the components, and deriving the property of the system from the component properties. Application of compositional reasoning to software systems requires establishing a compositional reasoning rule in the semantics of these systems,

⋆ This research was partially supported by NSF grant 010-3725.

proving the correctness of the rule, and implementing the rule. A rule is implemented when methods have been provided for discharging its premises which are usually verification of component properties, validity check of possible circular dependencies among component properties, and derivation of a system property from component properties.

Directly proving the correctness of compositional reasoning rules for software systems is often difficult. Software systems are usually modeled in specification languages such as Executable UML [10] and SDL [11], or coded in programming languages such as Java and C/C++. These languages are sufficiently complicated in syntax and semantics so that it is very difficult (if not infeasible) to directly prove for these languages that a compositional reasoning rule is sound. Additionally, such a language often has varying operational semantics. A formal semantics is only formulated for software systems specified in this language when these systems are to be translated into a model-checkable formalism and verified. On the other hand, proof and implementation of compositional reasoning rules for directly model-checkable formal semantics such as the semantics of Promela [12], SMV [13], and S/R [14] is often easier due to the formality and simplicity of these semantics. It is often the case that a set of compositional reasoning rules have already been proven and implemented for these semantics.

This paper defines, describes, and illustrates Translation-Based Compositional Reasoning (TBCR), an approach to application of compositional reasoning in the context of model checking software systems through model translation. This approach has two phases: (i) establishment of compositional reasoning rules in the semantics of software systems and correctness proof of the rules; (ii) application of the proven rules in model checking software systems. Given a translation from a software semantics to a directly model-checkable formal semantics, a compositional reasoning rule in the software semantics is established and proven for correctness as follows:

- The compositional reasoning rule is defined in the software semantics.
- The rule in the software semantics is mapped to an equivalent rule in the formal semantics based on the translation.
- The correctness proof of the rule is established based on the above mapping and on the correctness proof of the equivalent rule in the formal semantics.

Given a software system and a property to be checked on the system, the proven compositional reasoning rule in the software semantics is then applied as follows:

- The system is decomposed into components on the software semantics level.
- Premises of the rule are formulated in the software semantics. These premises are discharged by translating them to their counterparts in the formal semantics and discharging their counterparts in the formal semantics through reusing the implementation of the equivalent rule in the formal semantics.
- If these premises are successfully discharged, then it can be concluded on the software semantics level that the system has the property to be checked.

There has been a large body of research [4–9] (surveyed in [9]) on compositional reasoning in the formal methods community, which mostly focuses on developing compositional reasoning rules and proving their correctness. Our research, instead, focuses on effective application of compositional reasoning to software systems in the context of model checking these systems via model translation. Rationales for our approach are:

– Software systems, to be model checked, usually have to be translated into a directly model-checkable formalism.
– Formulation of and reasoning about the properties of software systems and their components are more naturally accomplished in the software semantics.
– Compositional reasoning rules have already been established, proven, and implemented for several directly model-checkable formalisms.

We have realized TBCR for a commonly used software semantics, the Asynchronous Interleaving Message-passing (AIM) semantics. In this realization, compositional reasoning rules in the AIM semantics are proven, implemented, and applied in the context of a translation from the AIM semantics to the ω-automaton semantics [15] using the I/O-automaton semantics [16] as an intermediate semantics. (We choose I/O-automata as the intermediate semantics to reuse a translation from the I/O-automaton semantics to the ω-automaton semantics, established by Kurshan, Merritt, Orda, and Sachs [17].) This realization has been applied in an integrated state space reduction framework [18] and in model checking of component-based software systems [19].

The balance of this paper is organized as follows. In Section 2, we give the preliminaries of the I/O-automaton semantics and the ω-automaton semantics. A realization of TBCR for the AIM semantics is defined and described in detail in Section 3. Two applications of the realization of TBCR for the AIM semantics and their case studies are presented in Section 4. We conclude in Section 5.

2 Preliminaries

2.1 I/O-automaton Semantics

The following definitions for I/O-automaton are from [17].

Definition 1. *An I/O automaton A is a quintuple $(\Sigma^A, S^A, I^A, \delta^A, R^A)$ where:*

– *the signature Σ^A is a triple $\Sigma^A = (\Sigma^A_{IN}, \Sigma^A_{OUT}, \Sigma^A_{INT})$, where Σ^A_{IN}, Σ^A_{OUT}, Σ^A_{INT} are pairwise disjoint finite sets of elements, called input, output, internal actions, respectively. We denote by $\Sigma^A_{EXT} = \Sigma^A_{IN} \cup \Sigma^A_{OUT}$ the set of external actions, by $\Sigma^A_{LOC} = \Sigma^A_{OUT} \cup \Sigma^A_{INT}$ the set of local actions, and we abuse notation, denoting by Σ^A also the set of all actions $\Sigma^A_{LOC} \cup \Sigma^A_{IN}$;*
– *S^A is a finite set of states;*
– *$I^A \subset S^A$ is a set of initial states;*
– *$\delta^A \subset S^A \times \Sigma^A \times S^A$ is a transition relation which is complete in the sense that for all $a \in \Sigma^A_{IN}$, $s \in S^A$ there exists $s' \in S^A$ with $(s, a, s') \in \delta^A$. For $a \in \Sigma^A_{LOC}$ and $s, s' \in S^A$ such that $(s, a, s') \in \delta^A$, we say that a is enabled at s and enables the transition (s, s'); Each element of δ^A is called a step of A;*
– *R^A is a partition of Σ^A_{LOC}, each element of which is termed a fairness constraint of A.*

Definition 2. *An execution of A is a finite string or infinite sequence of state-action pairs $((s_1, a_1), (s_2, a_2), \ldots)$, where $s_1 \in I^A$ and for all i, $s_i \in S^A$, $a_i \in \Sigma^A$ and $(s_i, a_i, s_{i+1}) \in \delta^A$.*

Definition 3. *An execution* $\mathbf{x}$ *of* A *is fair if, for all* $C \in R^A$*:*

- *if* $\mathbf{x}$ *is finite then no action in* C *is enabled in the final state in* $\mathbf{x}$*;*
- *if* $\mathbf{x}$ *is infinite then either some action in* C *occurs infinitely often in* x *or else infinitely many states in* $\mathbf{x}$ *have no enabled action which is in* C*.*

Definition 4. *Given a set* $\triangle \subset \Sigma^A$*, the projection of an execution* $\mathbf{x}= ((s_i, a_i))$ *of* A *onto* $\triangle$*, denoted by* $\Pi_\triangle(\mathbf{x})$*, is the subsequence of actions obtained by removing from the action sequence* (a_i) *all actions* $a_i \notin \triangle$*.*

Definition 5. *A behavior of* A *is the projection of a fair execution of* A *on the set* Σ^A_{EXT} *(i.e., the fair execution, with states and internal actions removed). The language* $\mathcal{L}(A)$ *of* A *is the set of all behaviors of* A*.*

Definition 6. *Of two I/O automata* A *and* B*, we say that* A *implements* B *(denoted by* $A \leq B$*) if, for* $\triangle = \Sigma^A_{EXT} \cap \Sigma^B_{EXT}$*,* $\triangle \neq \emptyset$*,* $\Pi_\triangle(\mathcal{L}(A)) \subset \Pi_\triangle(\mathcal{L}(B))$*.*

Definition 7. *For I/O automata* $A_1, A_2, \ldots, A_k$*, with respective pairwise disjoint sets of local actions, their parallel composition, denoted by* $A_1||A_2||\ldots||A_k$*, is an I/O automaton* A *defined as follows. The set of internal actions of* A *is the union of the respective sets of internal actions of the component automata, and likewise for the output actions; the input actions of* A *are the remaining actions of the components not thus accounted for. The set of states of* A*,* S^A*, is the Cartesian product of the component state sets, likewise for the initial states* I^A*. The transition relation* δ^A *is defined as follows: for* $s = (s_1, \ldots, s_k)$*,* $s' = (s'_1, \ldots, s'_k)$ *and* $a \in \Sigma^A$*,* $(s, a, s') \in \delta^A$ *if and only if for all* $i = 1, \ldots, k$*,* $(s_i, a, s'_i) \in \delta^{A^i}$ *or* $a \notin \Sigma^{A^i}$ *and* $s'_i = s_i$*.* R_A *is the union of the fairness partitions of the respective components.*

2.2 ω-automaton Semantics

We use the L-process model of ω-automaton semantics. Detailed specification of this model can be found in [15]. The concepts essential for understanding this paper are given below for the convenience of the reader.

Definition 8. *For an* L*-process,* ω*, its language,* $\mathcal{L}(\omega)$*, is the set of all infinite sequences accepted by* ω*.*

Definition 9. *For* L*-processes,* $\omega_1, \ldots, \omega_n$*, their synchronous parallel composition,* $\omega = \omega_1 \otimes \ldots \otimes \omega_n$*, is also an* L*-process and* $\mathcal{L}(\omega) = \cap\mathcal{L}(\omega_i)$*.*

Definition 10. *For* L*-processes,* $\omega_1, \ldots, \omega_n$*, their Cartesian sum,* $\omega = \omega_1 \oplus \ldots \oplus \omega_n$*, is also an* L*-process and* $\mathcal{L}(\omega) = \cup\mathcal{L}(\omega_i)$*.*

For a language, $\mathcal{L}$, let $\mathcal{CL}(\mathcal{L})$ denote the safety closure [20] of $\mathcal{L}$. [1]

[1] For a language $\mathcal{L}$ of sequences over a set of variables, V, the safety closure of $\mathcal{L}$, denoted by $\mathcal{CL}(\mathcal{L})$, is defined as the set of sequences over V where $x \in \mathcal{CL}(\mathcal{L})$ if and only if for all $j < |x|$ there exists y such that $x[0..j] : y$ belongs to $\mathcal{L}$ [8]. ($|x|$ denotes the length of x and $x : y$ denotes the concatenation of x and y where x and y are sequences over V.) In [15], $\mathcal{CL}(\mathcal{L})$ is termed as the smallest limit prefix-closed language that contains $\mathcal{L}$.

Definition 11. *The safety closure $CL^{\omega}(\omega)$ of an L-process ω is an L-process whose language is the safety closure of the language of ω, $\mathcal{L}(CL^{\omega}(\omega)) = \mathcal{CL}(\mathcal{L}(\omega))$.*

Given an L-process ω, $CL^{\omega}(\omega)$ can be derived from ω by computing the Strong Connected Components (SCCs) of the state graph of ω and for each SCC with an accepting state, marking every state of that SCC as accepting.

Under the ω-automaton semantics, model checking is reduced to checking L-process language containment. Suppose a system is modeled by the composition $\omega_1 \otimes \ldots \otimes \omega_n$ of L-processes, $\omega_1, \ldots, \omega_n$, and a property to be checked on the system is modeled by an L-processes, ω. The property holds on the system if and only if the language of $\omega_1 \otimes \ldots \otimes \omega_n$ is contained by the language of ω, $\mathcal{L}(\omega_1 \otimes \ldots \otimes \omega_n) \subset \mathcal{L}(\omega)$.

Definition 12. *Given two L-processes ω_1 and ω_2, ω_1 implements ω_2 (denoted by $\omega_1 \preceq \omega_2$) if $\mathcal{L}(\omega_1) \subset \mathcal{L}(\omega_2)$.*

3 Realization of TBCR for AIM Semantics

This section presents how TBCR is realized for the AIM semantics. First, we informally describe the AIM semantics. Then, we formalize the AIM semantics, which enables the establishment, correctness proof, implementation, and application of compositional reasoning rules. After that, we describe how a compositional reasoning rule for the AIM semantics is established. Then, we prove this rule based on a translation from the AIM semantics to the ω-automaton semantics using the I/O-automaton semantics as an intermediate semantics. Finally, we present the implementation of this rule through the translation from the AIM semantics to the ω-automaton semantics.

3.1 Informal Description of AIM Semantics

Under the AIM semantics, a system is a composition of processes that interact asynchronously via message-passing. Every process has a private message queue and locally defined variables. Behaviors of a process are captured by an extended Moore state model and each state in the state model may have an associated state action that is composed from executable statements such as an assignment statement, a messaging statement, and an "if" statement. At any given moment of a system execution, there is exactly one process that is executing either a state action or a state transition in a run-to-completion fashion.

3.2 Formalization of AIM Semantics

A state in the extended Moore state model of an AIM process represents a set of states in the state space of the process. A state action in the extended Moore state model represents multiple sequences of state transitions in the state transition structure of the process. To formally represent the extended Moore state model, we introduce a variable, *pc*, whose current value captures the current state in the Moore state model and the current position in the state action associated with the state. The message queue of the process is also formally represented by a variable, *queue*, whose domain includes all

possible message permutations that may appear in the queue. Under this representation of message queues, the execution of a messaging statement in a process modifies the *queue* variable of the receiver process. With the above representations, we formally define an AIM process.

Definition 13. *An AIM process, P, is a six-tuple, (S, I, M, E, T, F), where:*

- *S, the state space of P, is the Cartesian product of the domains of the variables defined in the process and the two additional variables, pc and queue.*
- *I is a set of initial states.*
- *M is a messaging interface which is a pair, (M^i, M^o), where M^i is the set of messages that P inputs and M^o is the set of messages that P outputs.*
- *E is a set of events each of which is a state transition of the Moore state model, or an executable statement (such as an assignment statement, a messaging statement sending a message defined in M^o, or an "if" statement), or a reception of a message defined in M^i. E_{LOC} is a subset of E including all state transitions and executable statements in E. E_{EXT} is a subset of E including all messaging statements and message receptions in E.*
- *T is a set of state transitions defined on S and E, each of which is of the form, $(s,\ e,\ s')$, where $s,\ s' \in S$ and $e \in E$.*
- *F is a partition of E_{LOC}. Each element of F is termed a fairness constraint.*

Definition 14. *An execution of P is a finite string or an infinite sequence of state-event pairs ($(s_0, e_0), (s_1, e_1), \ldots$) which conforms to the run-to-completion requirement (i.e., the action statements from a state action appear adjacently in the execution), where $s_0 \in I$ and for all i, $s_i \in S$, $e_i \in E$ and $(s_i,\ e_i,\ s_{i+1}) \in T$. Fair executions of P are defined analogously to fair executions of an I/O-automaton.*

Definition 15. *A behavior of P is the projection of a fair execution of P on E_{EXT} of P. The language of S, $\mathcal{L}(S)$, is the set of all behaviors of S.*

Definition 16. *Given two AIM processes P and Q, P implements Q (denoted by $P \models Q$) if for $\triangle = E_{EXT}(P) \cap E_{EXT}(Q)$ and $\triangle \neq \emptyset$, $\Pi_{\triangle}(\mathcal{L}(P)) \subset \Pi_{\triangle}(\mathcal{L}(Q))$.*

Definition 17. *The interleaving composition of a finite set of interacting AIM processes, P_0, P_1, ..., and P_n, denoted by $P_0 [] P_1 [] \ldots [] P_n$, is an AIM process, P, derived as follows. S is the Cartesian product of S_0, S_1, ..., and S_n. I is the Cartesian product of I_0, I_1, ..., and I_n. M^i includes the remaining messages in M_0^i, M_1^i, ..., and M_n^i that are not accounted for in the composition, and M^o is the union of M_0^o, M_1^o, ..., and M_n^o. E is the union of E_0, E_1, ..., and E_n. T is defined as follows: for $s = (s_0,\ s_1, \ldots,\ s_n)$, $s' = (s'_0,\ s'_1, \ldots,\ s'_n)$, and $e \in E$, $(s,\ e,\ s') \in T$ if and only if for all $i \in [0,\ n]$, $e \in E_i$ and $(s_i,\ e,\ s'_i)$ or $e \notin E_i$ and $s'_i = s_i$. F is the union of the fairness partitions of the respective components.*

In this formalized AIM semantics, a system, components of the system, and properties of the system and the components are all represented by processes.

3.3 Establishment of Compositional Reasoning Rules

We establish compositional reasoning rules for the AIM semantics by porting existing rules in directly model-checkable formal semantics to the AIM semantics. We have ported to the AIM semantics two rules that have already been established, proven, and implemented in the ω-automaton semantics, the rule proposed by Amla, Emerson, Namjoshi, and Trefler in [8], *Rule 1*, and the rule proposed by McMillan in [7]. Below we show how *Rule 1* is ported to the AIM semantics.

Rule 1 *For AIM processes P_1, P_2, and Q, to show that $P_1 [] P_2 \models Q$, find AIM processes Q_1 and Q_2 such that the following conditions are satisfied.*

C1: $P_1 [] Q_2 \models Q_1$ *and* $P_2 [] Q_1 \models Q_2$
C2: $Q_1 [] Q_2 \models Q$
C3: *Either* $P_1 [] CL^P(Q) \models (Q + Q_1 + Q_2)$ *or* $P_2 [] CL^P(Q) \models (Q + Q_1 + Q_2)$

Let $P_1 [] P_2$ denote a system composed from two components, P_1 and P_2. Q is a property to be checked on the system. Q_1 and Q_2 are properties of P_1 and P_2, respectively. Condition C1 checks if P_1 has the property, Q_1, assuming Q_2 holds on P_2, and if P_2 has the property, Q_2, assuming Q_1 holds on P_1. Condition C2 checks if Q can be derived from Q_1 and Q_2. Condition C3 conducts the validity check of circular dependencies between Q_1 and Q_2. (The counterpart of Rule 1 in the ω-automaton semantics, denoted by *Rule* 1^ω, is of the same form but with processes, $\models$, $[]$, CL^P, and $+$ replaced by their ω-automaton counterparts.)

To port compositional reasoning rules to the AIM semantics, additional semantics concepts may need to be introduced for the AIM semantics. In the case of Rule 1, the concepts of safety closure of an AIM process and sum of AIM processes were defined:

Definition 18. *For an AIM process, Q, the safety closure of Q, $CL^P(Q)$, is an AIM process whose language is the safety closure [20] of the language of Q, $\mathcal{L}(CL^P(Q)) = \mathcal{C}\mathcal{L}(\mathcal{L}(Q))$. ($CL^P(Q)$ can be derived from Q by removing the fairness constraints of Q.)*

Definition 19. *The Cartesian sum of AIM processes P and Q, denoted by $P + Q$, is the AIM process that behaves either as P or as Q and with the property of $\mathcal{L}(P + Q) = \mathcal{L}(P) \cup \mathcal{L}(Q)$.*

3.4 Proof via Semantics Translation

We first establish a translation from the AIM semantics to the ω-automaton semantics and then prove the soundness of Rule 1 based on the translation and the soundness proof of Rule 1^ω. To establish the translation from the AIM semantics to the ω-automaton semantics, we use the I/O-automaton semantics as an intermediate semantics.

Translation of AIM Processes to I/O-automata An AIM process, P, is translated to an I/O-automaton, A, through a two-step procedure. The first step maps semantic constructs of P to semantic constructs of A and the second step implements the run-to-completion requirement in A.
Step 1: Mapping semantic constructs

– The state space and the initial state set of P are mapped to the state space and the initial state set of A correspondingly, which is achieved by mapping the variables of P to the corresponding variables of A. (Note that the state space of an I/O automaton is also encoded by the domains of its variables.)
– Events of P are translated to actions of A as follows:
  - A state transition in the extended Moore state model of P is mapped to an internal action of A that simulates the state transition by modifying the variables, *pc* and *queue*, accordingly.
  - An assignment statement is mapped to an internal action that modifies the variable to be assigned by the assignment and the variable, *pc*.
  - An "if" statement is mapped to an internal action that modifies the variable, *pc*, to reflect the decision made in the "if" statement.
  - A messaging statement is mapped to an output action that is also an input action of the I/O-automaton corresponding to the receiver.
  - A message reception is mapped to an input action that modifies the variable, *queue*, and is also an output action of the sender I/O-automaton.
– Messages in the input (or output, respectively) interface of P are mapped to input (or output) actions of A.
– A state transition, $(s_P,\ e_P,\ s'_P)$, of P is mapped to a state transition, $(s_A,\ a_A,\ s'_A)$, of A where s_A, a_A, and s'_A are the corresponding translations of s_P, e_P, and s'_P as described above.

Step 2: Implementing run-to-completion requirement

– The I/O-automaton, A, resulting from Step 1 is extended with an additional boolean variable, *RtC*, and two output actions, *Enter* and *Leave*. The *Enter* action cannot be enabled unless the value of *RtC* is false.
– When A is composed with A', the I/O-automaton translation of another AIM process, P', the *Enter* and *Leave* actions of A are included by A' as input actions and vice versa.
– The transition relation of A is extended so that before A executes the first I/O-automaton action in the sequence of I/O-automaton actions corresponding to a state action of P, A executes the *Enter* action and after A executes the last I/O-automaton action in the sequence of I/O-automaton actions corresponding to a state action of P, A executes the *Leave* action. (A' is extended in the same way.)
– The transition relation of A' is extended so that as A executes the *Enter* action, A' sets its *RtC* to true and as A executes the *Leave* action, A' sets its *RtC* to false and vice versa.

Therefore, when a set of I/O-automata translated from AIM processes are ready to execute their *Enter* actions, only one of them can proceed, execute its *Enter* action, and get into the run-to-completion section. The automaton signals its leaving the run-to-completion section by executing its *Leave* action. We refer to the translation from an AIM process to its corresponding I/O-automaton as T_A^P.

Theorem 1. *Given an AIM process, $P = P_1 [] \ldots [] P_n$, and its I/O automaton translation, $A = T_A^P(P_1) || \ldots || T_A^P(P_n)$, for $\triangle = \Sigma_P^A$ where Σ_P^A is the set of external actions of A excluding all Enter and Leave actions and $\triangle \neq \emptyset$, $\mathcal{L}(P) = \Pi_\triangle \mathcal{L}(A)$.*

Proof of Theorem 1: By the construction of A from P, $\mathcal{L}(P) = \Pi_\triangle \mathcal{L}(A)$. □

Translation of AIM Processes to ω-automata Kurshan, Merritt, Orda, and Sachs [17] have established a translation from I/O-automata to ω-automata, T^A_ω, and also proved that the translation is linear-monotone with respect to language containment (shown in Theorem 2).

Theorem 2. *For two I/O-automata,* $A = A_1||\ldots||A_m$ *and* $B = B_1||\ldots||B_n$, $A \leq B \iff \mathcal{L}(T^A_\omega(A_1) \otimes \ldots \otimes \mathcal{L}(T^A_\omega(A_m)) \subset \mathcal{L}(T^A_\omega(B_1) \otimes \ldots \otimes \mathcal{L}(T^A_\omega(B_n))$.

Based on the translation from AIM processes to I/O-automata, T^P_A, and the translation from I/O-automata to ω-automata, T^A_ω, we constructed a translation from AIM processes to ω-automata, T^P_ω. For a given AIM process, P,

- P is first translated to an I/O-automaton $T^P_A(P)$;
- $T^P_A(P)$ is then translated to an ω-automaton $T^A_\omega(T^P_A(P))$.

We demonstrate with Theorem 3 that T^P_ω is also linear-monotone with respect to language containment.

Theorem 3. *For two AIM processes,* $P = P_1[]\ldots[]P_m$ *and* $Q = Q_1[]\ldots[]Q_n$, $P \models Q \iff \mathcal{L}(T^P_\omega(P_1) \otimes \ldots T^P_\omega(P_m)) \subset \mathcal{L}(T^P_\omega(Q_1) \otimes \ldots T^P_\omega(Q_n))$.

Proof of Theorem 3: Follows directly from Theorem 1 and Theorem 2. □

Lemma 1. *For an AIM process P,* $CL^\omega(T^P_\omega(P)) \preceq T^P_\omega(CL^P(P))$.

Proof of Lemma 1:

$\Rightarrow$ {Definition 18, Definition 16}
$P \models CL^P(P)$
$\Rightarrow$ {Theorem 3}
$\mathcal{L}(T^P_\omega(P)) \subset \mathcal{L}(T^P_\omega(CL^P(P)))$
$\Rightarrow$ {Monotonicity of language closure}
$\mathcal{CL}(\mathcal{L}(T^P_\omega(P))) \subset \mathcal{CL}(\mathcal{L}(T^P_\omega(CL^P(P))))$
$\Rightarrow$ {Definition 11}
$\mathcal{L}(CL^\omega(T^P_\omega(P))) \subset \mathcal{CL}(\mathcal{L}(T^P_\omega(CL^P(P))))$
$\Rightarrow$ {A safety property is the safety closure of itself.}
$\mathcal{L}(CL^\omega(T^P_\omega(P))) \subset \mathcal{L}(T^P_\omega(CL^P(P)))$
$\Rightarrow$ {Definition 12}
$CL^\omega(T^P_\omega(P)) \preceq T^P_\omega(CL^P(P))$

□

Lemma 2. *For AIM processes* $P_1, \ldots, P_n$, $T^P_\omega(P_1 + \ldots + P_n) \preceq T^P_\omega(P_1) \oplus \ldots \oplus T^P_\omega(P_n)$.

Proof of Lemma 2: Follows directly from Definition 19, Theorem 3, Definition 10, and Definition 12. □

Theorem 4. *Rule 1 is sound for arbitrary AIM processes, P_1, P_2, and Q.*

Proof Sketch of Theorem 4: Suppose Conditions C1, C2, and C3 hold on P_1, P_2, and Q. Due to Theorem 3, Lemma 1, and Lemma 2, the counterparts of Conditions C1, C2, and C3 in the ω-automaton semantics hold on $T_\omega^P(P_1)$, $T_\omega^P(P_2)$, and $T_\omega^P(Q)$. Therefore, by Rule 1^ω (the counterpart of Rule 1 in the ω-automaton semantics), $T_\omega^P(P_1) \otimes T_\omega^P(P_2) \preceq T_\omega^P(Q)$. By Theorem 3, we conclude that $P_1 [] P_2 \models Q$. (Detailed proof of this theorem can be found in the appendix.) □

3.5 Implementation and Application of Rule 1 through Model Translation

TBCR suggests that a compositional reasoning rule in the AIM semantics be implemented based on the translation from the AIM semantics to the ω-automaton semantics and by reusing the implementation of its equivalent rule in the ω-automaton semantics. We first introduce an implementation of the AIM-to-ω-automaton translation and an implementation of Rule 1^ω (the ω-automaton semantics counterpart of Rule 1) in the ω-automaton semantics. We then discuss how Rule 1 is implemented and applied.

Translation from xUML to S/R xUML [10] is an executable dialect of UML whose semantics conforms to the AIM semantics given in this paper. S/R [14] is an automaton language whose semantics conforms to the ω-automaton semantics. In previous research [18][21], we have implemented a translator from xUML to S/R. Given a system modeled in xUML and a property specified in an xUML level logic, the design and the property are translated to an S/R model and an S/R query. The S/R query is checked on the S/R model by the COSPAN [14] model checker. The property holds on the system if and only if the S/R query is successfully verified on the S/R model. As shown in Figure 1, the xUML-to-S/R translation syntactically translates an xUML

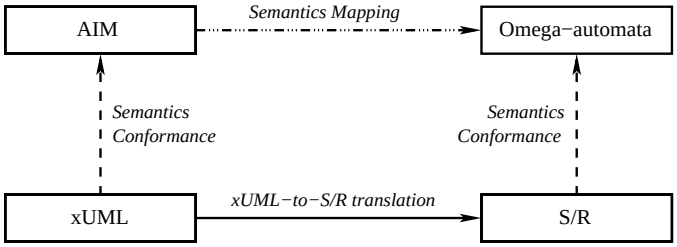

Fig. 1. xUML-to-S/R translation implements semantics mapping from AIM to ω-automata.

model into S/R, which also implements the semantics mapping from the AIM semantics to the ω-automaton semantics.

Existing Implementation of Rule 1^ω in S/R Rule 1^ω has been implemented in S/R [8]. Since in S/R, systems, components, assumptions, and properties are all modeled as ω-automata which can be trivially composed, verification of component properties (Condition C1) and derivation of a system property from component properties (Condition C2) are discharged in the same way as a property is checked on a system. Validation of circular dependencies (Condition C3) additionally requires construction of the safety closure of an ω-automaton (which has been discussed in Section 2.2).

Implementation and Application of Rule 1 in xUML The xUML-to-S/R translator requires that an xUML model to be translated specify a closed system. To support Rule 1, the translator is extended to allow a closed system formed by a component of a system and its assumptions on the rest of the system (i.e. properties that the component assumes the rest of the system to have). The extension is simplified by the fact that in S/R, systems, components, assumptions, and properties to be checked are all modeled as ω-automata which can be trivially composed. Based on the implementation of Rule 1^ω in S/R and the extended xUML-to-S/R translator, compositional reasoning using Rule 1 is applied in model checking software systems modeled in xUML as follows:

- Given a system modeled in xUML and a property to be checked, the system is decomposed on the xUML level and premises of Rule 1 are formulated in xUML.
- These premises are discharged by translating them to their counterparts in S/R using the extended xUML-to-S/R translator and discharging their counterparts using the implementation of Rule 1^ω in S/R.

Correct application of Rule 1 then depends on the correctness of the translation from xUML to S/R and the correctness of the implementation of Rule 1^ω in S/R.

4 Applications

We presents two major applications of the realization of TBCR for the AIM semantics.

4.1 Application in Integrated State Space Reduction Framework

In previous research [18], we presented an integrated state space reduction framework for model checking executable object-oriented software system designs. This framework is presented for system designs modeled in xUML, but can also be readily used to structure integrated state space reduction for other representations. As shown in Figure 2, the framework structures the application of state space reduction algorithms into three phases, the user-driven state space reduction phase, the xUML-to-S/R translation phase, and the S/R model checking phase. Different algorithms are applied in each phase and the application of an algorithm may span multiple phases. (In Figure 2, an algorithm is only associated with the phase in which it is initiated.) Interactions among these algorithms are utilized to maximize aggregate effect of state space reduction.

TBCR is one of the most powerful state space reduction algorithms applied in this framework. Its application spans across all the three phases:

- In the user-driven state space reduction phase, a system (or a large component of the system) specified in xUML is decomposed into components and properties of the components are specified. Premises of Rule 1, verification of component properties, derivation of system properties from component properties, and validation of possible circular dependencies among component properties, are all formulated (on the AIM semantics level) as verification sub-tasks generated in the decomposition.
- These sub-tasks are either recursively reduced (on the AIM semantics level) with user-driven state space reduction into simpler sub-tasks, or translated into the S/R automaton language through the xUML-to-S/R translation phase and discharged (on the ω-automaton level) in the S/R model checking phase.

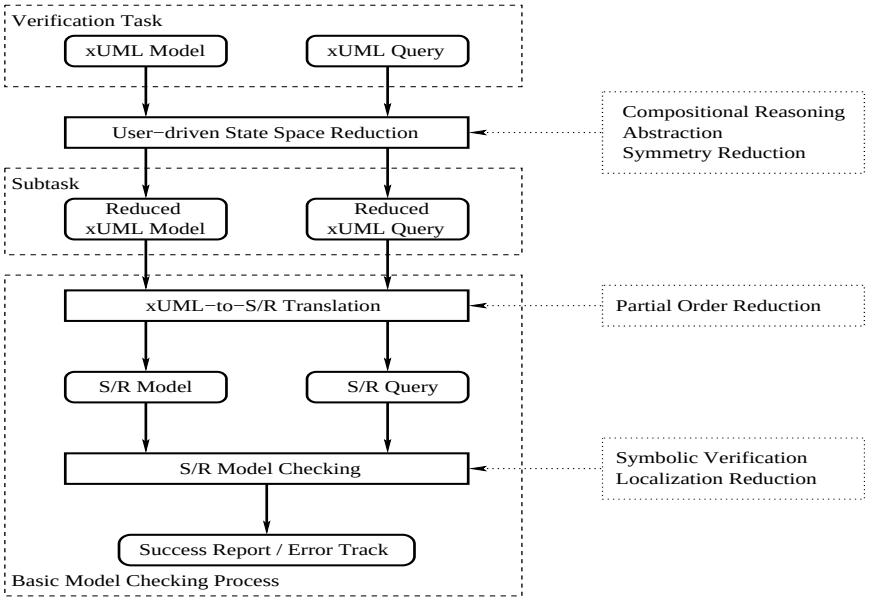

Fig. 2. Reduction hierarchy of integrated state space reduction framework

The general framework has been instantiated for the domain of distributed transaction systems by utilizing domain-specific design patterns. The instantiation has been applied in model checking an online ticket sale system. Figure 3 shows the decomposi-

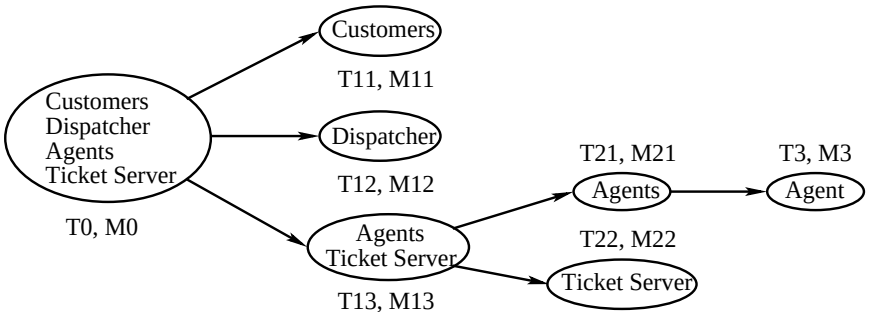

Fig. 3. Decomposition of online ticket sale system

tion of the system generated in checking an availability property, P: *After a request from a customer is received, a reply is eventually sent back to the customer*. In Figure 3, *M0* denotes the complete model that consists of the *customers*, the *dispatcher*, the *agents*, and the *ticket server* while *M11*, *M12*, *M13*, *M21*, *M22*, and *M3* denote the submodels of *M0* derived in the decomposition. *T0* is a verification task defined on *M0*: Checking the property, P, on the model, *M0*. *T0* is decomposed into a set of verification subtasks, *T11*, *T12*, and *T13*, which check properties of *M11*, *M12*, and *M13* locally. These properties of *M11*, *M12*, and *M13* are specified according to the decomposition. (Derivation of P from these properties of *M11*, *M12*, and *M13* and validation of possible circular dependencies between these properties are also verification sub-tasks, however, such subtasks are not shown in Figure 3 for the sake of conciseness.) A verification subtask,

for instance, *T13*, may be further decomposed. To discharge *T0*, only the verification subtasks on the leaf nodes of the decomposition tree must be discharged. A verification subtask is discharged by translating the corresponding submodel with its assumptions on other submodels into S/R and checking the corresponding properties on the resulting S/R model. Detailed discussion of this case study can be found in [18].

4.2 Application in Integration of Model Checking into Component-Based Development of Software Systems

Overview In [19], we defined, described, and applied an approach to integration of model checking into component-based development (CBD) of software systems, which is based on compositional reasoning and can be summarized as follows:

- As a software component is built, temporal properties of the component are formulated, verified, and then packaged with the component.
- Selecting a component for reuse considers not only its functionality but also its temporal properties.
- Verification of properties of a composed component reuses verified properties of its sub-components and is based on compositional reasoning.

Traditional applications of compositional reasoning take a top-down approach: To check properties of a system, the system is decomposed into modules recursively in a top-down fashion. (The application of TBCR in the integrated state space reduction framework, presented in Section 4.1, follows the top-down approach.) This integration of model checking into CBD combines the top-down application of TBCR with the bottom-up component composition process of CBD and discharge premises of a compositional reasoning rule by reusing previous component verification efforts as possible. Using Rule 1 as the compositional reasoning rule, the combination is conducted as follows:

- A property of a component is defined together with assumptions on the environment of the component and is verified on the component under these assumptions. When the component is reused in the composition of a larger component, the property is *enabled* if the environment assumptions made in its verification hold on other components in the composition and/or the environment of the composed component.
- As a primitive component (a component built from "scratch" and not composed from other components) is verified, its properties are directly model checked on the executable representation of the component such as its executable design model.
- As a composed component is verified, premises of Rule 1 are discharged as follows:
  Condition C1: Verification of sub-component properties is reused from previous verification efforts.
  Condition C2: A property of the composed component is derived by being verified on an abstraction of the component, which is constructed from environment assumptions of the component and verified properties of its sub-components. A verified sub-component property is included in the abstraction if it is *enabled* in the composition and related to the property of the composed component according to the cone-of-influence analysis.
  Condition C3: Validation of circular dependencies among sub-component properties is executed to decide if a sub-component property is properly *enabled*.

In our implementation of this integration of model checking into CBD, the executable representations of components are specified in xUML. Therefore, formulation of and reasoning about the component properties are conducted in the AIM semantics. Following the TBCR approach, premises of Rule 1 are formulated in the AIM semantics and checked by translating them to their counterparts in S/R and then model checking their counterparts on the ω-automaton level.

Case Study The integration of model checking into CBD has been applied to improve reliability of run-time images of TinyOS [22], a component-based run-time environment for networked sensors. In this case study, we discuss how the integration is applied in verifying a run-time image of TinyOS, the Sensor-to-Network component, which is composed from the Sensor component and the Network component. The Sensor (or Network, respectively) component outputs messages of the types, *Output* and *Done_Ack* (or *Data_Ack* and *Sent*), and inputs messages of the types, *OP_Ack* and *Done* (or *Data* and *Sent_Ack*). Figure 4 shows an abstracted communication diagram of the

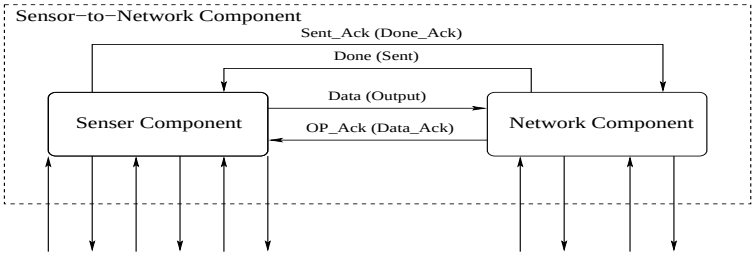

Fig. 4. Sensor-to-Network component

Sensor-to-Network component, where an annotation of the form of "Input message type (Output message type)" denotes that an output message type of a component is mapped to an input message type of the other component. In Figure 4, the arrows coming in and going out of the dashed box denote interrupts from the hardware platform and their corresponding replies. For the sake of conciseness, we omit the assumptions of the component on the hardware platform in the following discussion.

The goal of this case study is to check whether the Sensor-to-Network component has the following property (denoted by Q): *The component transmits sensor readings on physical network repeatedly*. The formal specification of Q is shown in Figure 5. *RFM.Pending* is a variable defined in the Network component. Setting and then clearing this variable indicate a transmission over the physical network. Q_1 and Q_2 are two properties formulated on the Sensor component. Q_1 asserts that the Sensor component outputs sensor readings repeatedly and Q_2 asserts that the Sensor component properly handles its output hand-shakes. Q_3 and Q_4 are two properties formulated on the Network component. Q_3 asserts that the Network component transmits on physical network repeatedly if it inputs repeatedly and Q_4 asserts that the component properly handles its input hand-shakes.

Condition C1: Verification of sub-component properties In previous verification studies, Q_1 and Q_2 have been verified on the Sensor component by assuming Q_4 holds on its environment. (Q_4, when used as an assumption of Q_1 and Q_2, is formulated on

Properties of Sensor-to-Network Component:
Property Q
Repeatedly (RFM.Pending);
Repeatedly (Not RFM.Pending);

Properties of Sensor Component:
Property Q_1
Repeatedly (Output);

Property Q_2
After (Output) **Never** (Output) **UntilAfter** (OP_Ack);
After (Done) **Eventually** (Done_Ack);
Never (Done_Ack) **UntilAfter** (Done);
After (Done_Ack) **Never** (Done_Ack) **UntilAfter**(Done);

Properties of Network Component:
Property Q_3
IfRepeatedly (Data) **Repeatedly** (RFM.Pending);
IfRepeatedly (Data) **Repeatedly** (Not RFM.Pending);

Property Q_4
After (Data) **Eventually**(Data_Ack);
Never (Data_Ack) **UntilAfter** (Data);
After (Data_Ack) **Never** (Data_Ack) **UntilAfter** (Data);
After (Sent) **Never** (Sent) **UntilAfter** (Sent_Ack);

Fig. 5. Properties of TinyOS components

input and output message types of the Sensor component.) Q_3 and Q_4 have been verified on the Network component by assuming Q_2 holds on its environment. Since the Sensor and Network components are both primitive components, the verification was conducted by translating the xUML design models of the two components into S/R and model checking on the S/R level.

Condition C3: Validation of circular dependencies An abstraction of the Sensor-to-Network component was constructed for verifying Q. Verification of Q on the abstraction failed since the abstraction cannot include Q_1, Q_2, Q_3, and Q_4 due to the circular dependency between Q_2 and Q_4. The circular dependency between Q_2 and Q_4 was validated by checking whether one of the following conditions holds:

- $Sensor \,[]\, CL^P(Q_2 [] Q_4) \models (Q_2 [] Q_4 + Q_2 + Q_4)$;
- $Network \,[]\, CL^P(Q_2 [] Q_4) \models (Q_2 [] Q_4 + Q_2 + Q_4)$.

These two verification tasks were discharged by translating them into S/R and model checking on the S/R level. Both conditions hold in this case.

Condition C2: Derivation of properties of composed component The abstraction was then refined by including Q_1 and Q_3 since the circular dependencies between Q_2 and Q_4 have been validated. Q was successfully verified on the refined abstraction by translating the abstraction into S/R and model checking on the S/R level.

Since conditions C1, C2, and C3 have been successfully discharged, it can then be concluded with Rule 1 that Q holds on the Sensor-to-Network component.

5 Conclusions

TBCR is a simple and effective approach to application of compositional reasoning in the context of model checking software systems via model translation. It simplifies the correctness proof of compositional reasoning rules in software semantics and reuses existing proofs and implementations of compositional reasoning rules in directly model-checkable semantics. The feasibility and effectiveness of TBCR has been demonstrated by its realization for the AIM semantics and by two applications of this realization.

Acknowledgment

We gratefully acknowledge Nina Amla and Nancy MacMahon for their generous help. We also thank the anonymous referees for their valuable suggestions.

References

1. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. Logic of Programs Workshop (1981)
2. Quielle, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. 5th International Symposium on Programming (1982)
3. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. The MIT Press (1999)
4. Pnueli, A.: In transition from global to modular reasoning about programs. Logics and Models of Concurrent Systems (1985)
5. Alur, R., Henzinger, T.: Reactive modules. LICS (1996)
6. Abadi, M., Lamport, L.: Conjoining specifications. TOPLAS (1995)
7. McMillan, K.L.: A methodology for hardware verification using compositional model checking. Cadence TR (1999)
8. Amla, N., Emerson, E.A., Namjoshi, K.S., Trefler, R.: Assume-guarantee based compositional reasoning for synchronous timing diagrams. TACAS (2001)
9. de Rover, W.P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Non-compositional Proof Methods. Cambridge Univ. Press (2001)
10. Mellor, S.J., Balcer, M.J.: Executable UML: A Foundation for Model Driven Architecture. Addison Wesley (2002)
11. ITU: ITU-T Recommendation Z.100 (03/93) - Specification and Description Language (SDL). ITU (1993)
12. Holzmann, G.: Design and Validation of Computer Protocols. Prentice Hall (1991)
13. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers (1993)
14. Hardin, R.H., Har'El, Z., Kurshan, R.P.: COSPAN. CAV (1996)
15. Kurshan, R.P.: Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton University Press (1994)
16. Lynch, N.: Distributed Algorithms. Morgan Kaufmann Publishers (1996)
17. Kurshan, R.P., Merritt, M., Orda, A., Sachs, S.R.: Modeling asynchrony with a synchronous model. Formal Methods in System Design 15(3) (1999)
18. Xie, F., Browne, J.C.: Integrated state space reduction for model checking executable object-oriented software system designs. FASE (2002)
19. Xie, F., Browne, J.C.: Verified systems by composition from verified components. ESEC/FSE (2003)
20. Alpern, B., Schneider, F.: Defining liveness. Information Processing Letters **21** (1985)

21. Xie, F., Levin, V., Browne, J.C.: ObjectCheck: a model checking tool for executable object-oriented software system designs. FASE (2002)
22. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for networked sensors. ASPLOS-IX (2000)

Appendix: Detailed Proof of Theorem 4

Proof of Theorem 4:

$$
\begin{aligned}
& P_1 [] Q_2 \models Q_1 \\
\Rightarrow\ & \{\text{Theorem 3}\} \\
& T_\omega^P(P_1) \otimes T_\omega^P(Q_2) \preceq T_\omega^P(Q_1) \qquad (1)
\end{aligned}
$$

$$
\begin{aligned}
& P_2 [] Q_1 \models Q_2 \\
\Rightarrow\ & \{\text{Theorem 3}\} \\
& T_\omega^P(P_2) \otimes T_\omega^P(Q_1) \preceq T_\omega^P(Q_2) \qquad (2)
\end{aligned}
$$

$$
\begin{aligned}
& Q_1 [] Q_2 \models Q \\
\Rightarrow\ & \{\text{Theorem 3}\} \\
& T_\omega^P(Q_1) \otimes T_\omega^P(Q_2) \preceq T_\omega^P(Q) \qquad (3)
\end{aligned}
$$

$$
\begin{aligned}
& P_1 [] CL^P(Q) \models (Q + Q_1 + Q_2) \\
\Rightarrow\ & \{\text{Theorem 3}\} \\
& T_\omega^P(P_1) \otimes T_\omega^P(CL^P(T)) \preceq (T_\omega^P(Q + Q_1 + Q_2)) \\
\Rightarrow\ & \{\text{Lemma 1, Lemma 2}\} \\
& T_\omega^P(P_1) \otimes CL^\omega(T_\omega^P(T)) \preceq (T_\omega^P(Q) \oplus T_\omega^P(Q_1) \oplus T_\omega^P(Q_2)) \qquad (4)
\end{aligned}
$$

$$
\begin{aligned}
& P_2 [] CL^P(Q) \models (Q + Q_1 + Q_2) \\
\Rightarrow\ & \{\text{Theorem 3}\} \\
& T_\omega^P(P_2) \otimes T_\omega^P(CL^P(Q)) \preceq (T_\omega^P(Q + Q_1 + Q_2)) \\
\Rightarrow\ & \{\text{Lemma 1, Lemma 2}\} \\
& T_\omega^P(P_2) \otimes CL^\omega(T_\omega^P(Q)) \preceq (T_\omega^P(Q) \oplus T_\omega^P(Q_1) \oplus T_\omega^P(Q_2)) \qquad (5)
\end{aligned}
$$

$$
\begin{aligned}
& \{(1), (2), (3), (4), (5)\} \\
\Rightarrow\ & \{\text{Rule } 1^\omega\} \\
& T_\omega^P(P_1) \otimes T_\omega^P(P_2) \preceq T_\omega^P(Q) \\
\Rightarrow\ & \{\text{Theorem 3}\} \\
& P_1 [] P_2 \models Q \qquad (6)
\end{aligned}
$$

□