JCSE Journal of Computing Science and Engineering

Regular Paper

Journal of Computing Science and Engineering,
Vol. 20, No. 1, March 2026, pp. 1-15

SBC Process Algebra for SysML Model Inconsistency Checking

Steve W. Haga* **and William S. Chao**
Department of Computer Science and Engineering, National Sun Yat-sen University, Kaohsiung, Taiwan
stevewhaga@yahoo.com, architectchao@gmail.com

Wei-Ming Ma
Department of Information Management, Cheng Shiu University, Kaohsiung, Taiwan
k3666@ggcloud.csu.edu.tw

Abstract
This paper proposes a unified approach for checking the consistency of SysML models by integrating structural and behavioral information into a single framework, the interaction transition graph (ITG). The ITG is a finite state machine representation derived from process algebra, with a specific focus on structure-behavior coalescence (SBC), combining both structure and behavior into a single format. Being formal, efficient, and scalable, this approach addresses the challenges of inconsistency that arise when dealing with complex systems and multiple SysML diagrams. The ITG is designed to be a human-readable representation that provides the user with a consistent view of the full system. The paper also presents a software tool that converts the process algebra equations for the various SysML diagrams into the ITG representation. The use of the ITG for consistency checking is also demonstrated.

Category: Software Engineering

Keywords: Model-based systems engineering; Consistency checking; Interaction transition graph; Structure-behavior coalescence; Process algebra

I. INTRODUCTION

SysML defines eight standard diagrams. Five focus on structural information: block definition diagram (BDD), internal block diagram (IBD), package diagram (PkD), parametric diagram (PmD), and use case diagram (UCD). Two focus on behavioral information: state machine diagram (SMD) and activity diagram (AD). And one focuses on both, the sequence diagram (SD). In simple terms, the difference between these is that structural diagrams define static aspects such as components and their interactions, while behavioral diagrams define interactions that dynamically occur based on the system state. Among these diagrams, the UCD relies more upon human-readable text and it lacks the details found in the other diagrams. Consequently, we will hence forth refer to only seven SysML diagrams.

Each of the remaining seven diagrams present a different view of the system structure or behavior. Consistency checking is needed to detect if these different views have any contradicting information about either the system's structure or behavior.

Open Access http://dx.doi.org/10.5626/JCSE.2026.20.1.1 http://jcse.kiise.org

This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/licenses/by-nc/4.0/) which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

Received 4 May 2025; **Revised** 27 January 2026; **Accepted** 11 March 2026
*Corresponding Author

Copyright © 2026. The Korean Institute of Information Scientists and Engineers pISSN: 1976-4677 eISSN: 2093-8020

To facilitate consistency analysis, many representations have been proposed to express the information from various diagrams into some unified syntax, such as colored Petri nets, description logic, or institutions. Consistency checking tends to apply an ad-hoc process of searching for individual causes of potential inconsistency, one-by-one. While some representations are logical or mathematical descriptions suitable for software analysis, they may not be suitable for human examination. Many of these representations focus only on behavioral information and may not cover all possible behaviors. The desirability of a unified representation for consistency checking is indicated by the proposing of so many different prior representations. The core reason why consistency issues remain a challenge is that the various diagrams share partial common information, but represent that information in very different ways and at different levels of granularity.

In this paper, we propose the interaction transition graph (ITG) as a representation for all seven SysML structural and behavioral diagrams. The ITG is a finite state machine (FSM), which is based upon process algebra and represents a labeled transition system (LTS).

The proposed LTS offers several advantages. First, the process algebra description offers a high-level overview, which helps both in debugging inconsistencies and in providing a new view of the system design. This human-readable semantic description can help users in verifying the consistency of the human-language parts of UCD description. Second, the SD large amounts of information (both structural and behavioral), and the above-mentioned overview diagram that SBC provides has a general correspondence to the SD, greatly facilitating the representation of the combines structural and behavioral information of the SD within our coalesced format. Third, the process algebra allows for a formal procedure for generating the LTS representation of each diagram. Fourth, the ITG is shown to be able to represent all information from the various structure and behavioral diagrams into one structure-behavior coalescece (SBC) format. Fifth, the process algebra connects additional information to the individual SysML diagrams. The SMD, for example, cannot, by nature, have transition edges for weak transitions (although a complex of states can together represent such transitions), but the process algebra which created the SMD can both: 1) indicate which transitions are, in fact weak, and 2) automatically generate a complex of states for that weak behavior, if desired.

SBC is a system model that has been used for many years as a model for system behavior. SBC views a system's structure and its behavior as being so closely linked that it is difficult to understand one without considering the other. So, SBC proposes an LTS representation of both a system's structure and behavior. More recently, SBC has been adapted to model some of the diagrams shared by SysML and UML. In this new adaptation, the SBC-based LTS representation becomes the formal semantic description of the SysML models. In this present paper, we demonstrate SBC's ability to model all of a system's structural and behavioral information within one diagram, by providing techniques to translate each of SysML's diagrams into a common LTS format that captures both structural and behavioral information.

Several advantages come from the SBC approach of merging structure and behavior into a single unified representation. First, it assists in finding potential conflicts in situations where different structures might be able to perform the same behavior. Second, it assists in debugging inconsistencies, as the representation can identify not just the inconsistent behavior, but also the structures producing the conflict. Third, various SysML diagrams have both structural and behavioral information, particularly the sequence diagram. By represent the behavior in an SBC format, the representation is closer to the original models it is derived from, which helps in human analysis of consistency problems.

In previous works, we have presented process algebra representations for three behavioral diagrams: the SD [1], the SMD [2], and the AD [3], and have presented a combined representation of the SD and the SMD for consistency checking [4]. In this present work, we introduce structural information into the LTS representation to create the ITG. Compared to the purely-behavioral ITG in our previous works, the ITG adds edge label information regarding the identity of the sender, receiver, and parameters.

With this addition, the new ITG provides a single consistent view of structural and behavioral information in one diagram. Moreover, this paper presents a procedure for consistency checking ITGs, and procedures for extracting specific structural or behavioral information from the ITG. A software tool is also presented to demonstrate the applicability of the approach.

In the following sections we will begin with a literature review in Section II. This will consider various approaches to structural and behavior consistency checking. It concludes with a discussion of our previous works that this current work is built upon. Next, In Section III, we discuss our SBC representation of structural and behavior al information. The core idea is that comparing the different diagrams is assisted by developing a common format for expressing all of the information in the diagrams. This requires Section III to present a precise formal semantics for our representation of this information. Section IV then applies this formalism to each specific SysML diagram, including an illustrative example to clarify the means of representing each diagram. Section V then presents our demonstration tool for implementing these concepts. Finally, Section VI concludes.

II. LITERATURE REVIEW

Two general approaches are possible to perform

consistency checking. The first is to construct a large set of known potential consistency problems, and then to implement a procedure to check for each of these. Such an ad-hoc procedure is hinted at in [5], which lists the many causes of inconsistencies. It is also likely to be the approach used in many industrial tools, given that it can be used without developing a full theory of model design. The problem for the approach is that cases can be missed and errors can be made in the code to check for the long list of possible problems.

The second approach is to create some descriptive format that allows information from different diagrams to be described with the same syntax. Once this is achieved, a consistency solver appropriate to that format is often used. The advantage of such approaches is that they offer a more formal procedure, which should streamline the verification procedure (so as to reduce analysis time) and also to reduce the likelihood of bugs or missing inconsistency cases.

A. Structural Consistency Checking

SysML's approach to system specification is partly derived from the computer science concept of objects. In SysML, blocks encapsulate both structural and behavioral characteristics, including properties, operations, and constraints. This description approach promotes modularity, allowing system engineers to manage complexity through hierarchical decomposition and clear component interaction definitions.

Structural consistency checking involves ensuring that both the blocks representing system components and their specified interactions are correctly defined and are consistent across the various diagrams and model views. Modern tools for structural consistency checking utilize techniques such as model transformation into a common format, constraint checking of predefined rules to verifying that they meet design requirements, and formal verification model checking. These techniques are integrated into modeling environments and toolchains like the Eclipse Modeling Framework (EMF) [6] and Model-Based Systems Engineering (MBSE) [7] platforms.

Our proposed approach does not fundamentally change these concepts. Our main contribution in this regard is a table format that provides an organized view of all of the structural information, which can simplify the approach to identifying structural inconsistencies. Our table also has the advantage of tying these structural details back to the specific diagrams and areas of the diagrams that produced these issues. In addition, we demonstrate a tool to extract this information.

B. Behavioral Consistency Checking

Most previous works on consistency checking between diagrams focus on behavioral analysis, as structural consistency problems are relatively easier to identify. Concerning behavioral analysis, methods usually focus on extracting semantic meaning into a shared structured format such as Z-notation [8] or institutions [9]. Once these facts are transformed into a unified format, software performs consistency checks. However, these methods may not account for all possible inconsistency cases.

Our approach also converts into a shared format, the ITG, which is a LTS based on process algebra. In our case, this common representation format serves as a complete metamodel for all potentially-inconsistent SysML diagram behaviors, preserving all necessary semantic meaning to ensure a thorough consistency verification.

Although no previous works have specifically sought to create a metamodel for all SysML behavioral diagrams, some do convert a diagram into a state machine (or state-machine-like) representation, so as to offer a more formalized and precise semantic meaning. Regarding the SMD, it is a specialized form of state machine already, although some works do simplify the SMD representation into either a reduced form of state machine [10] or a colored Petri net (which could in principle then be converted to a state machine) [11]. The AD also bears a correspondence to a state machine (in approximately the same way a flow chart can be transformed into state machine).

As for the SD, these also have been converted to interaction automata (IA) [9]. These IA are similar to our LTS, but they are not derived from process algebra, so they do not impose a simple design to the IA. In [9], the focus is on simple model checking (e.g., finding illegal behaviors), rather than on exhaustively searching the SD's IA for any inconsistencies with the SMD.

A single unified transition system (TS) for the SD, SMD and AD is also shown in [8]. If the TS is built from an SMD, then it becomes very related to our ITG for the SMD. However, the primary goal of [8] is not generate the TS, but to use the TS to generate model checker information into the format of Z-notation, so that use-case scenarios can be evaluated. Although, [8] is focused on Z-notation, rather than creating a meta-model, the approach does involve converting an SD into a state machine. It therefore creates a semantic description that could have been extended into a meta model (though such an extension is not considered in [8]). The complexities of handling non-simple SDs are also not considered by [8]. Moreover, the performance of consistency checking would not benefit from optimization, because (as with the other related works presented above) no structure is imposed upon the meta-model, unlike our process-algebra-based approach.

Although prior works do not explore using their derived state machines as meta-models to compare to other diagrams, we can imagine that next step. Indeed, a table on construction-based techniques in [12] demonstrates a similar line of reasoning.

As for our current work, it is aided by our previous works [1-3] that presented methods for converting, respectively, the SD, SMD, and AD into LTS representations. In [1], the SD is shown to have similarities to the overview diagram that we create by parsing the process algebra equation that describes the system. This is because a parse tree can be represented as nested boxes that are very similar to the structure of the SD. In [2], the SMD is shown to have the properties of an LTS. By presenting a procedure to convert process algebra into an LTS, we therefore achieve something similar to converting a simplified SD into an SMD. In [3], the AD is discussed, with a process algebra grammar that allows its behavior to be converted into an SMD.

The SMD and AD contain similar behavioral information, although one presents the information in a state machine format and the other in a flow-chart-like format. Thus in [4], we introduced consistency checking between the SD and SMD. With an LTS, consistency checking can be achieved by an exhaustive search for behavioral equivalence between the ITGs created from an SD and an SMD. Using their shared process algebra descriptions, however, reduces the search space, as the algebraic descriptions of the SD and SMD can be directly compared, with LTS analyzers, such as [13].

III. THE SBC REPRESENTATION OF STRUCTURAL AND BEHAVIORAL INFORMATION

To create a specification of interactions that covers both structural and behavioral information, we use the syntax of operation call-and-return signatures [14]. In this specification, behavior is defined by methods, and structure is defined by the details used to invoke those methods (such as the parameters' names, the caller's name and called method's name and the name of the object providing that method). Given that previous works for representing SysML diagrams have tended to separate the structure from the behavior, such signatures have not previously been formalized for SysML diagrams.

A. Specifying Interactions with Operation Call/Return Signatures

Because we achieve the combination of structural information with the behavior it implements by introducing the concept of call/return signatures, their details need to be clearly described. To understand them, first consider that system behavior is often visualized using the paradigm of object-oriented programming where the system components function as objects and their interactions are represented as methods.

Although this is a useful framework we note that real system behavior may not correspond to the call-and-return structure of programming languages. For example, in Fig. 1(a), Object D returns directly to Object B, without revisiting Object C. The solution for modeling SysML is to allow for the separation of calls from returns, which is the purpose of the operation call-and-return signature representation.

Fig. 1(b) presents how the seven interactions of the sequence diagram in Fig. 1(a) are represented in this syntax. As shown in Fig. 1(b), the syntax allows for call signatures, return signatures, and call-and- return signatures. Call-and-return signatures can be decomposed, so: $call_sig_{CD}$ = D.h(In: x; Out: y) = $call_sig_{CD}$ + ret_sig_{DC} = D.h(In: x) + D.h(Out: y). The reason to allow this decomposition is illustrated in Fig. 1(a), where the call to B.f() must be separated from its return, since there are other interactions that must be specified between the call and return.

Making use of the concept of operation signatures, we now provide precise definitions for three key terms, as they are used in the SBC approach of this paper: messages,

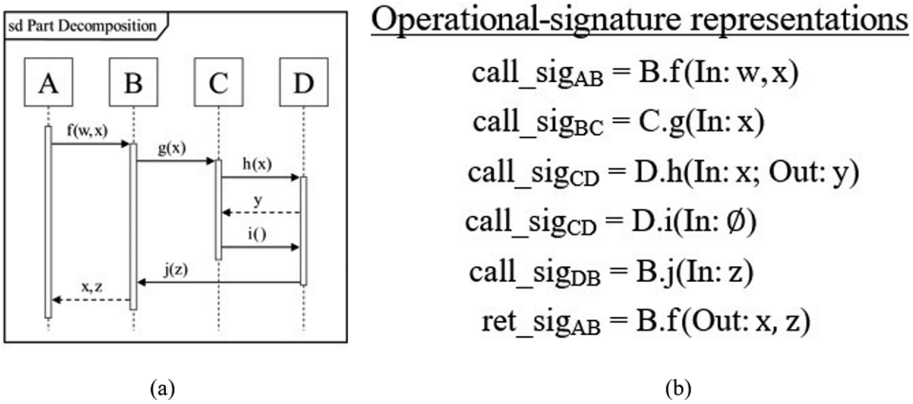


Fig. 1. An illustration of the operation call-and-return signature representation. (a) A sample sequence diagram with four interacting objects. (b) The operation call-and-return signature representation. Particularly note that Object D returns directly to Object B, without revisiting Object C.

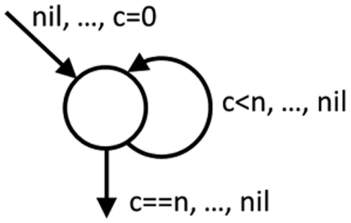


Fig. 2. How two prefix fields describe loops within an ITG. This figure shows a single state that belongs to some larger ITG. The transitions in an ITG are labeled with all of the information of the guarded interaction specification. In this figure, however, only the two fields of guard condition and code snippet are shown.

(1) s ::= •
(2) s ::= iss **strict** s
(3) iss ::= i **seq** iss
(4) iss ::= i
(5) s ::= (s_1 **alt** s_2)
(6) s ::= (s_1 **par** s_2)
(7) s ::= LOOP (ITG)
(8) s ::= PARSTEP (ITG) **strict** s
(9) s ::= ref A

Fig. 3. The SBC grammar for SysML.

interactions, and guarded interactions. These terms require clear definitions, because different SysML diagrams use similar words to refer to different things. When a user of sequence diagrams refers to an "interaction," it only means behavioral information. Also, a SD "message" may refer to either a message-sending event or a message-receiving event, but a SMD "message" is a combined message sending-and-receiving event. To clarify these issues, we therefore define the following:

- Message: The information passed between objects; it is located in the arguments of the call and return signatures.
- Interaction: The message information, along with the identification information of the caller and callee objects. In our implementation, this information is represented by the operation call-and-return signature format.
- Guarded interaction: In this paper, the prefix *i* always refers to a guarded interaction. This is an interaction augmented with fields for a guard condition and any code snippet needed to control that guard condition. To clarify how a code snippet might control a condition, consider a looping behavior where an interaction iterates *n* times. As Fig. 2 shows, an initial code snippet (c = 0) is needed to achieve this looping behavior.

B. A Process Algebra Grammar for SysML

Building upon the grammars proposed in our previous works for the SD, SMD, and AD [1-3], we now present a combined grammar for all the SysML diagrams of interest. In the following, "•" indicates a termination state, "*i*" indicates a guarded interaction that functions as the prefix in the process algebra specification, and "*iss*" indicates an interaction-specifying state (or sequence of interaction-specifying states) (Fig. 3).

In this grammar, Rule 1 ensures that every process description terminates. Rules 2 and 3 describe strict and weak sequences, respectively. Rule 4 is used to define the prefix of a single interaction. Rules 5 and 6 describe alternative and parallel behaviors, respectively. As for the final three rules, these are more complex and will be discussed below. But as for the first 6 rules, these are derived from the rules given in [1]. In [1], these rules were formalized by defining transition steps that allow any process algebra equation using these rules to be converted into an LTS of that implements the precise behavior specified in that equation.

Although the full details for Rules 1–6 can be found in [1], the general concept of the conversion to an LTS can be briefly provided here. As for Rule 1, it defines the termination state, "•", that occurs in the LTS. Rule 2 defines strict sequences. In the corresponding LTS, a strict sequence of interactions becomes a linear sequence of states. Rule 3 describes weak sequences, where the sending of a message is a separate event that the receiving of that message. There is a sense of sequential order, because messages must be sent before they are received; but additional messages are allowed to be sent before the first message is received, so it is only a weak sense of order (i.e., it is a partial order). As for the transition rule to obtain the corresponding LTS, a weakly-sequenced chain of interactions becomes a triangular mesh of potential state transitions (such as the upper-triangle mesh depicted on the right-side of Fig. 4(c)). The triangle shape arises because of how Rules 3 and 4 relate to each other. It bears noting that weak sequences are only defined for sequence diagrams (although we extend the specification of diagrams such as the SMD to also cover weak sequences, as described in the discussion of Fig. 4(c)). Next, Rule 5 describes alternative behaviors, and the transition rule to create the LTS for this is to define edges to two separate next-states. As for Rule 6, it describes parallel behavior, and LTSs allow for parallelism between states, so this is directly modeled.

As for the final three rules, these warrant closer discussion, as follows.

Rule 7 allows the user to describe looping behavior through a provided ITG. This rule follows our previous works (e.g., [3]), but is a departure from most process algebra grammars, which typically model looping either by unrolling the loop or away, or by using recursion. The purpose of Rule 7 is to allow system engineers to have a more natural way to describe complex looping

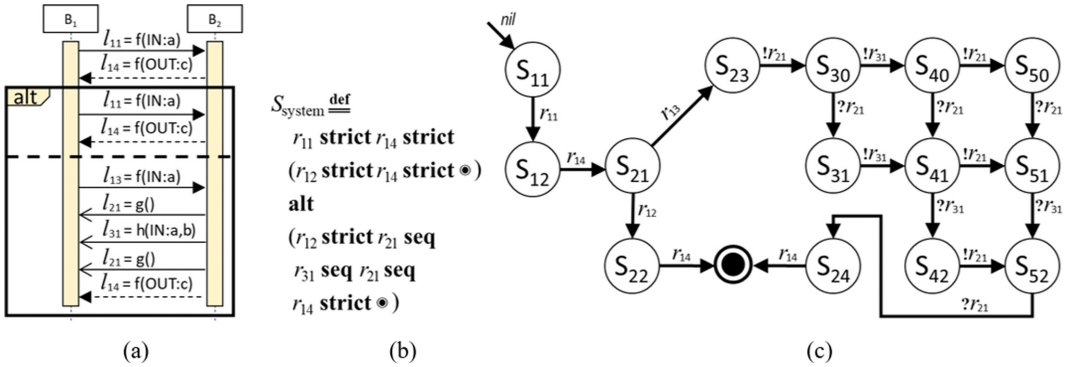


Fig. 4. Constructing a message sending-and-receiving transition graph (MSRETG) from a sequence diagram (SD). (a) A sample SD for a system that is consistent with the AD of Fig. 5(a). The SD is labeled with operational signatures, l_x. The three arrows from B_1 are calls to B_2's operation *f*, while the three dashed arrows are the corresponding return values. Also, this SD has two alternatives. Along the lower path, B_2 calls B_1's operations (*g* and *h*). These calls are indicated as not returning values and as being weakly sequenced (i.e., they have open arrow heads). (b) The process algebra equation in the syntax of [1], where weak and strict sequences are labeled seq and strict, respectively. (c) The MSRETG, as defined in [1]. Some prefix labels are marked with a "!" or "?", which is a common notation for separately referencing the sending and the receiving of weakly sequenced events (i.e., in this case, l_{21}, l_{31}, and l_{21}). The order of such events is indeterminant, so the MSRETG is a lattice of branching paths, as detailed in [1].

behaviors.

Rule 8 allows the user to describe a partial ITG to model a step that must complete before the system is allowed to proceed (i.e., the "**strict** *s*" part of Rule 8). The ITG used in Rule 8 is termed partial, because it does not end in the termination state as is standard for process algebra, but rather ends in a continuation state [3]. Rule 8 is helpful for simplifying the specification of decision / merge node pairs without the need to use Rule 9. More importantly, it is needed for specifying the fork / join node pairs of concurrent systems.

In introducing Rule 8, it is necessary to define how it can be translated into the full-system ITG. An extensive discussion can be found in [3], but briefly, it involves using the guards and code snippets at entry and exit to the PARSTEP region. To explain this behavior, consider, as an example, a system that is defined as:

Sys = i_1 **strict** PARSTEP (ITG_P) **strict** i_2 **strict** •, with $ITG_P = i_a$ **par** i_b. Then we convert it to Sys′ = i_1' **strict** (i_a' **strict** •) *par*(i_b' **strict** i_2' **strict** •), where

$i_1 = gc_1, \ldots, cs_1 \Rightarrow i_1' = gc_1, \ldots, (cs_1; c = 0)$
$i_2 = gc_2, \ldots cs_2 \Rightarrow i_2' = gc_1$ &* $c == 2, \ldots, cs_2$
$i_a = gc_a, \ldots, cs_a \Rightarrow i_a' = gc_a, \ldots, (cs_a; c \mathrel{+}= 1)$
$i_b = gc_b, \ldots, cs_b \Rightarrow i_b' = gc_b, \ldots, (cs_b; c \mathrel{+}= 1)$

In the above conversion, behavior is not allowed to proceed to i_2 until both concurrent processes i_a and i_b complete—which means that the join node behavior is achieved.

Rule 9 is standard in process algebra. This rule allows an alias for a predefined or imported process, promoting reusability and modularity within the system. Referencing an existing process avoids redundancy and helps to build complex system descriptions from smaller, predefined components.

C. Formal Definitions of Interactions and ITG Transitions

Based on the above discussion of interactions, the following definitions and relational specifications can be defined:

Interaction: A system's interactions can be defined as $I = (\Xi, N, \Gamma, \Lambda, \Theta_{IN}, \Theta_{OUT}, IR)$, where:

Ξ - A finite set of caller agents (objects or actors that require the interactions);

N - The nature of the interaction (a call, return, or call-and-return);

Γ - A finite set of callee agents (objects providing the interactions);

Λ - The operation provided by the callee agent;

Θ_{IN} - The message (i.e., input parameters) sent to the callee agent;

Θ_{OUT} - The response (i.e., output parameters) returned by the callee agent;

IR - An interaction relation, $IR \subseteq \Xi \times N \times \Gamma \times \Lambda \times \Theta_{IN} \times \Theta_{OUT}$, where $(\xi, n, \gamma, \lambda, \theta_{IN}, \theta_{OUT}) \in IR$.

Prefix (also referred to as a guarded interaction): The prefixes of a system can be defined as $PX = (C, IR, \Pi, PR)$, where:

C - A finite set of optional guard conditions;

IR - The interaction relation;

Π - A finite set of optional code snippets;

PR - A prefix relation, $PR \subseteq C \times IR \times \Pi$, where $(c, e, \pi) \in PR$.

Interaction transition graph: An SBC ITG can be defined as $ITG = (\Psi, PR, ITGR, s_0)$, where:

Ψ - A finite set of system states

PR - The prefix relation

$ITGR$- The ITG relation, $ITGR \subseteq \Psi \times PR$, where $(s, i) \in ITGR$;

s_0 - An initial dummy state that transitions to the true initial state of the system, where the interaction on this transition is ε, a spontaneous transition. A code snippet can also be attached to this transition to allow initialization of the system. For systems starting in multiple concurrent states, parallel transitions out of s_0 are used.

Combining the above, we obtain: $ITGR \subseteq \Psi \times C \times \Xi \times N \times \Gamma \times \Lambda \times \Theta_{IN} \times \Theta_{OUT} \times \Pi$ Here, the edges that define transitions between states are labeled with eight fields that precisely define the interaction occurring between components at that particular transition, in terms of both structure and behavior. The inclusion of structural information on these edge labels is a key feature of the SBC approach, ensuring that behavior is always considered in the context of the system's architecture.

D. The Sufficiency of the SBC Approach for Consistency Checking

The sufficiency of the SBC approach follows directly from several axioms:

(1) Process algebra is known to be able to represent any behavior of any system.

(2) Any equation written in process algebra is known to be able to be converted into a behaviorally-equivalent LTS, with the individual system behaviors being specified by the labels on the transitions between states. Moreover, the LTS provides the formalism for process algebra. This is because each individual potential walk through the LTS directly corresponds to one of the system's set of potential traces.

(3) By definition, any interaction that occurs within a system must be either an interaction between system components or else an interaction between a system component and an actor. Said in different words, this merely observes that system behaviors can only occur if the structure of the system enables those behaviors.

Combining Axioms 2 and 3, we find that: all of a system's behaviors are specified on the labels of the transitions of the LTS, and all of those behaviors are performed by known structural elements. By annotating these labels with the corresponding structural information, SBC is achieved. Therefore, we have:

(4) SBC process algebra is able to represent all of the structural and behavioral information found within any of the SysML diagrams. And this leads to the conclusion:

(5) If an inconsistency exists between two SysML diagrams, then that inconsistency will also exist in the SBC process algebra's LTSs for those two diagrams. This means that the bisimilarity tests that have long been applied by LTS solvers can discover any behavioral inconsistencies between diagrams, for cases that have constrained loops and code snippets.

The above constraint caveat is necessary because unconstrained code snippets would allow for Turing completeness, which is well known to create difficult to analyze behavior. If, however, the ITG's code snippets are limited to controlling loop bounds, and if the loops themselves are ordinary, with a constant loop index stride, then the behavior can be properly analyzed and understood by the LTS solver. We also require that conditional test expression be identical, because allowing different expressions also exposes the unknowable property of generic Turing machines.

Looking back to Statement (5), while the truth of this statement is straightforward to establish, it leaves open the more difficult problem of how exactly to identify such inconsistencies between two SBC process algebra equations.

The issue is that different diagrams may have different levels of granularity. The SD can use the IGNORE combined fragment to ignore specific interactions. Also, the SD can define weakly-sequenced events where the sending and the receiving of a message are treated as separate events; when that same behavior is being specified as a SMD, the engineer might choose to reduce the number of states by treating the message sending-and-receiving as a single event. Such issues complicate the bisimilarity testing procedure employed by LTS solvers for validating consistency between model descriptions.

Our LTS solver handles this issue by dynamically creating pseudo-states that merge any transitions that are present in one LTS but are ignored by the other LTS it is being tested against. In other words, these pseudo-states that the LTS solver dynamically creates allow the two LTSs to be analyzed at the same granularity. To give a more formalized description of the effect, note that the full behavior of a system is formally defined as the set of all valid traces that can occur when the system is run. Two system descriptions are equivalent if they create the same set of traces. By introducing these pseudo-states, our LTS solver is, in effect, creating subtraces where those interactions that are ignored have been removed from all of the traces in an LTS's set of traces. Thus, formally stated, two models are consistent if the set of their reduced traces are identical.

IV. THE INTERACTION TRANSITION GRAPH AND THE GUARDED INTERACTION TABLE

A. Representing Individual Diagrams

Three of our previous works have each presented separate LTSs for representing three behavioral diagrams [1-3]. These will be discussed first.

The AD has similarities to a flow chart, with activities being defined inside of nodes. As Fig. 5(a) shows, both the calling ($\rightarrow$) and returning-from ($\leftarrow$) a function are activities, but there are two exceptions: sometimes they can be combined into one activity ($\rightleftarrows$), and sometimes a function does not return (e.g., $g()$ and $h()$). The procedure

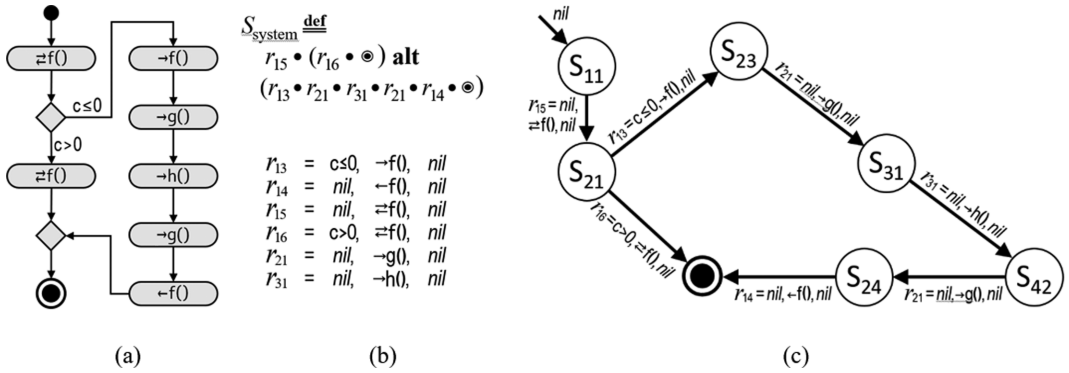


Fig. 5. Constructing an action transition graph (ATG) from an activity diagram (AD). (a) A sample AD. Here, the activities in the elliptical nodes are calls and returns from various functions. (b) This system's process algebra equation, in the AD-targeted syntax of [3]. The r_{xx} terms are called prefixes and their specific index number labels are arbitrarily chosen. As in [3], these prefixes are triplets that specify: an optional guard condition, an action, and an optional code snippet. Fig. 5 has no code snippets (they are not a focus of this present paper), and the only guard is for deciding on the alternative path to take from S_{11}. (c) The corresponding ATG. The states' specific index number labels are arbitrarily chosen (The "*nil*" label into S_{00} is an empty code snippet).

of [3] is then applied to derive a process algebra equation, so as to produce a specific LTS format that was termed the action transition graph (ATG). As can be seen, the ATG is an LTS that formally defines the behavior of the system represented by the AD. Additionally, in passing, note that Fig. 5(b) demonstrates that the prefixes in by SysML already is an LTS, so Fig. 5(c) is already a legal SMD. That is: Fig. 5(c) is not only the ATG for the given AD, but it is also the SMD to describe that same system (although, in this case, it would not be called an ATG, but an event transition graph (ETG) [2]. In fact, the reason that the AD was converted into an ATG was so that it could be compared with other diagrams, such as the SMD [4]. The key point of [2], which targeted SMDs, was not the creating of an LTS, but the using of process algebra equations (such as Fig. 4(b)) for defining the behavior of the SMD.

The SD presented in Fig. 4(a) is consistent with the behavior of the AD shown in Fig. 5(a), but the SD also contains two kinds of additional information. One addition is structural information (regarding which blocks provide the various function operations and about which parameters are used by those operations). The LTS in Fig. 5(c) (which follows the ideas of [1]) only represents the behavioral aspects. A second addition is the ability to specify a weak sequence of events, for describing situations where events might reach the receiver in a different order than they were sent. Due to the need to model this additional information, the LTS in Fig. 5(c) is called a message sending-and-receiving event transition graph (MSRETG) [1].

Although the SD has additional information compared to the AD, it also lacks some of the AD's information. Consider the alt operation shown in Fig. 4(a). Notice that the figure does not specify what guard condition is that is used to choose between the alternatives. In contrast, the AD in Fig. 5(a) indicates conditions of $c > 0$ and $c \leq 0$. As Fig. 5(b) shows, the r_{xx} prefixes do contain a field for guard conditions; therefore, the r_{xx} prefixes of additional information specified by the system designer, such as through an annotation attached to the alt operator. Therefore, the l_{xx} operational signatures of Fig. 4(a) lack information present in the r_{xx} prefixes of Fig. 4(b). This fact explains why Fig. 4(a) contains two instances of l_{11}, but Fig. 4(b) represents these separately as r_{11} and r_{12}; r_{11} and r_{12} differ in the condition field.

The issue of weak sequences merits further discussion. While the SD defines weakly-sequentialized events, the AD and SMD do not. This difference can be interpreted in two different ways: (1) that the SMD and AD diagrams are intended to represent higher-granularity behavior than the SD, and therefore they do not specify weak sequencing, or (2) that the representing of weak sequencing is desirable for SMDs and ADs, but these diagrams are unfortunately just not well suited to the task. For example, the transition from one state to another in a state machine is an atomic operation—there is no concept of a second transition starting before the first transition has finished. In this view, it is not correct to say that the SMD cannot represent weak sequences, but only to say that the representation of such information will be complex; Fig. 4(c) is, in fact, a state machine that is able to represent weak sequencing information, by means of creating many intermediary states. If an engineer did this intricate work by hand, mistakes producing inconsistencies could occur. As for our ITG, it is a state machine, but it is tied to a process algebra with weak sequence syntax. So, the designer can simply choose which way (simplify to strict or leave as weak) to handle the expansion of weak sequences in the process algebra when autogenerating the SMD.

When performing consistency checking, the only way for information to be inconsistent is for that information to differ between diagrams. Information that is only present in one of the diagrams can therefore not have an inconsistency with the other diagram. In this regard, the

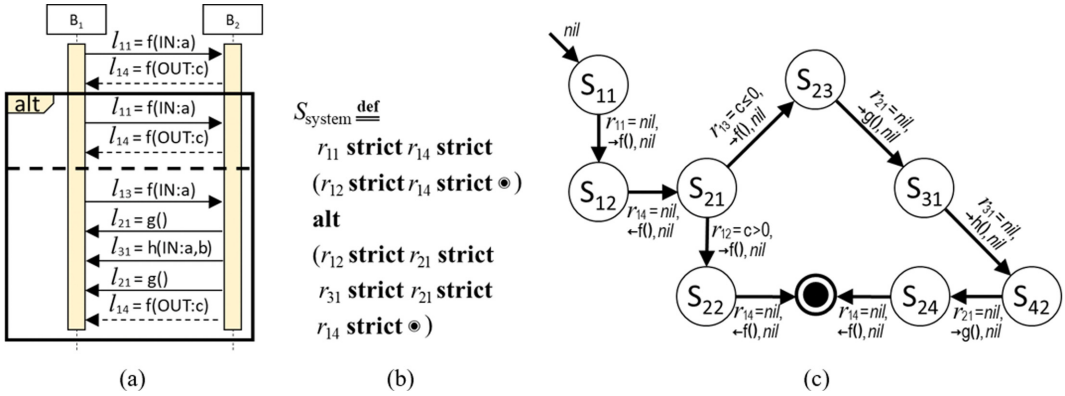


Fig. 6. Constructing an ITG from a SD. (a) As noted in the text and in [4], weak sequences cannot cause inconsistencies with other diagram types, so the SD is reduced to strict sequences (i.e., the hollow arrow heads of Fig. 4(a) are now made solid). (b) The new process algebra equation. (c) The ITG of [4]. Note that both S_{12} and S_{22} only have one in-edge and one out-edge (r_{11} and r_{12}). These correspond to *f*'s operations to call (l_{11}) and return (l_{14}). Thus, r_{11} and r_{12} meet the conditions (described in the text below) for merging.

AD and SMD do not distinguish between strict and weak sequencing, so the SD's weak sequences cannot produce inconsistencies with those diagrams. It is therefore helpful to convert all weak sequences in the SD into strict ones (this is legal because the order of events described by a strict sequence is one of the valid possible outcomes of a weak sequence description) [4].

Fig. 6 presents the SD with weak sequences converted to strict. When comparing to the process algebra equations of Figs. 5(b) and 6(b), the **strict** operator of Fig. 6(b) is equivalent to the "•" operator of Fig. 4(b) (the reason for the difference is that Figs. 5 and 6 follow the different-but-equivalent syntaxes presented in [3] and [1], respectively). Comparing the graphs of Figs. 5(c) and 6(c), Fig. 6(c) has extra states, because all three of *f*()'s calls (r_{11}, r_{12}, r_{13}) have corresponding return edges (r_{14} in each case). In Fig. 4(c), however, two of these call-return pairs (r_{11}/r_{14} and r_{12}/r_{14}) have each been merged into the action of calling-and-returning (r_{15} and r_{16}). This merging is not always possible, and so Fig. 4(c) still separated r_{13} from its instance of r_{14}. These two could not be merged, because other actions needed to occur between these events.

One characteristic of the SMD (and also of the AD) is that they do not specify weak sequencing—in contrast to the sequence diagram that will be discussed next. By their nature, SMDs represent full transition from one state to the next. Therefore, SMDs do not define weak transitions, although the behavior can be modeled by dividing the sending and receiving into separate events and adding extra states. A series of weak transitions will create many extra internal states, as Fig. 4(c) illustrates. A benefit of our approach of using process algebra to generate the SMD is that it allows simple generation of those internal states, if they are desired by the system engineer. Alternatively, if the engineer wants a simpler SMD, weak sequences in the process algebra can be treated as strict sequences, during the SMD generation process. This is legal because the strict sequence order is a valid possibility for a series of weak sequences.

B. Creating the ITG as a Unified LTS

In [4], the only way to check the consistency of the SD against the SMD was to require that the SMD not merge calls and returns, which, we now note, was an unnecessary burden upon the system engineer. We therefore propose the representation illustrated in Fig. 7(b), which merges two returns (states S_{12} and S_{22}) into their corresponding calls (states S_{11} and S_{21}). With the states now being consistent between Figs. 5(c) and 7(b), bisimilarity analysis can be used to check for consistency between them.

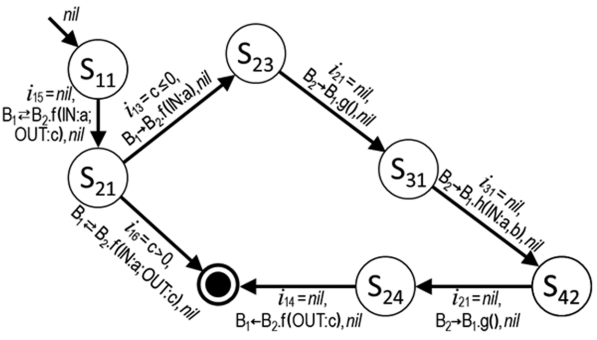
Another new feature of Fig. 7(b) is that the edges have i_{xx} labels instead of the r_{xx} labels used previously. The "*i*" stands for interaction, and the changing of label name represents that the proposed ITG contains new information not found in the earlier graphs—namely, structural information. The edges in Fig. 6(c) indicate what action occurs on an edge transition, but the edges in Fig. 7(b) additionally indicate how that action is implemented. For example, compare Fig. 6(c)'s prefix (r_{31} = *nil*, → *h*() *nil*) to Fig. 7(b)'s corresponding interaction (i_{31} = *nil*, $B_2 \rightarrow B_1$.h(IN:a,b), *nil*). The interaction has structural information about which component is providing the service, and what parameters are provided.

Fig. 7(a) presents these i_{xx} values in tabular form. As shown, an interaction is an 8-tuple (C, Ξ, N, Γ, Λ, θ_{IN}, θ_{OUT}, Π), where some fields have structural information and others have behavioral information. On the right-side of the table, the sites where these interactions occurred are also noted; the purpose of this noting will be discussed in the next section.

The introduction of structural information into the edges of the LTS represents a shift in thinking about systems. It is the concept of SBC, which recognizes the ways in which these two system components are interconnected. If one wishes to only see the behavior, one merely blanks

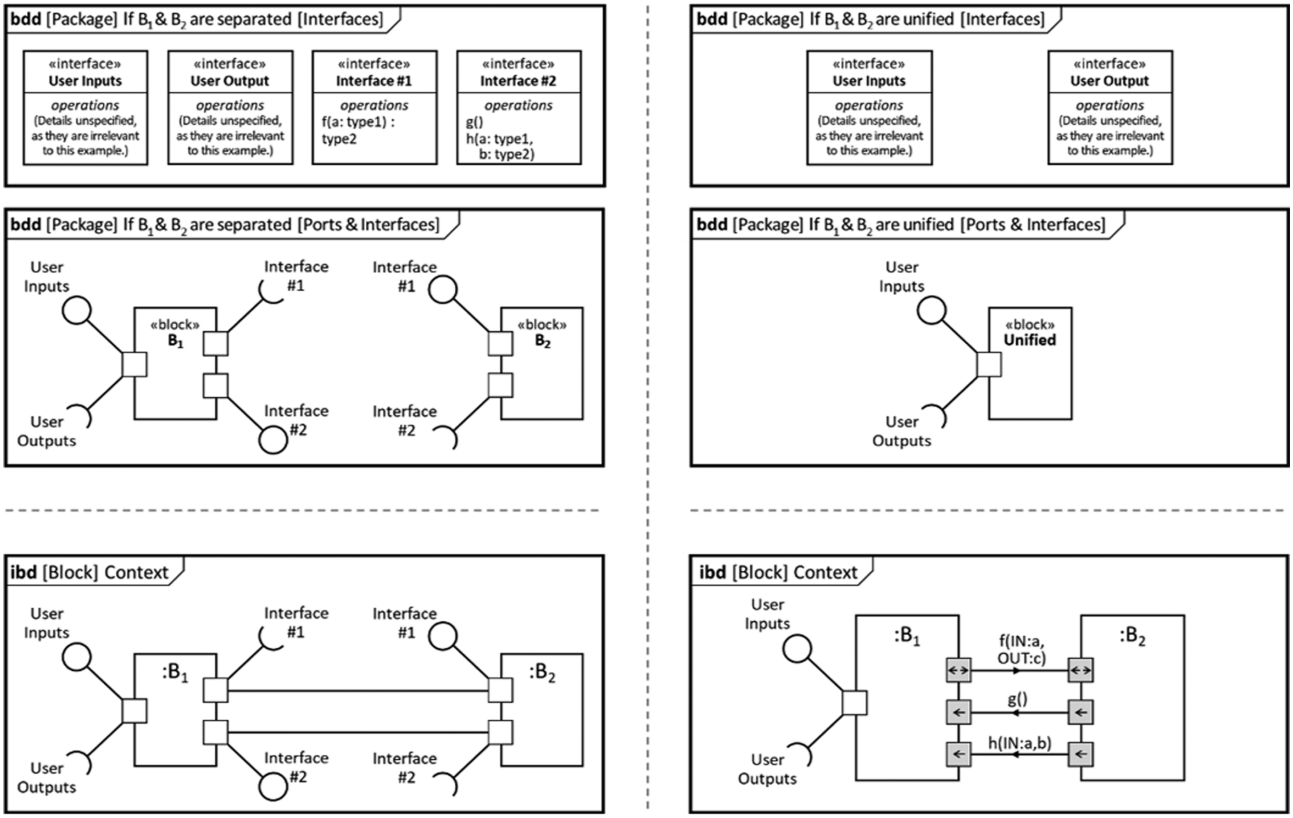
I	C	Ξ	N	Γ	Λ	θ_{IN}	θ_{OUT}	Π	Sites
i_{13}	c≤0	B_1	Call	B_2	f	a	*nil*	*nil*	S_{21}
i_{14}	*nil*	B_1	Ret	B_2	f	*nil*	c	*nil*	S_{24}
i_{15}	c>0	B_1	Call	B_2	f	a	c	*nil*	S_{11}
i_{16}	*nil*	B_1	Call	B_2	f	a	c	*nil*	S_{21}
i_{21}	*nil*	B_2	Call	B_1	g	*nil*	*nil*	*nil*	S_{23}, S_{42}
i_{31}	*nil*	B_2	Call	B_1	h	a,b	*nil*	*nil*	S_{31}

(a)



(b)

Fig. 7. The proposed GIT and ITG. (a) shows the guarded interaction table (GIT). The GIT lists each interaction's 8 fields, augmented with the states where they occur. In the table, grey columns are behavioral and white columns are structural. (b) shows the simplification of Fig. 6(c), with S_{12} and S_{22} being merged into other states and removed.



(a) (b)

Fig. 8. Two alternative forms for the BDD and IBD. (a) The two BDDs and one IBD for the situation where B_1 and B_2 are in separate blocks. Here, because B_1 and B_2 are separated, there is internal structure—so the IBD is very similar to the BDD for "Ports & Interfaces." (b) If B_1 and B_2 are instead united into one block. In this case the IBD has internal structure, and it is the BDD for "Ports & Interfaces" that becomes simple.

out the structural fields from the interactions' 8-tuples. When one wishes to see the structure, it is represented in a way that directly ties it to the behavior.

Concerning the structural data seen in the extended interaction table (GIT) of Fig. 7(a), it all derived from the SD of Fig. 6(a). The GIT is referred to as an extended table, as it extends SysML's definition of an interaction, to include behavioral information in the *C* and *Π* fields. The GIT is not needed for consistency checking with the AD and SMD behavioral diagrams, but it will be need for other SysML diagrams that do have structural information, as will now be discussed.

The block decision diagram (BDD) and internal block diagram (IBD) are SysML structural diagrams. As Fig. 8 shows, the designer has a choice of which components to consider as belonging to the same block. The designer could represent an entire complex system as single block, could group some components into blocks for specific subsystems, or could put each component into its own block; the specific purpose of any given system would determine the best choice. Since Fig. 8's system is a generic example, either the choice of block grouping is possible.

Comparing Fig. 8(a) and 8(b), the purpose of showing two different options is that they are different ways of presenting the same structural information, but partitioned differently. In Fig. 8(a), the top BDD ("Interfaces") has the (Λ, θ_{IN}, θ_{OUT}) information, the other BDD ("Ports & Interfaces") has the (Γ) information, and the IBD has the (Ξ, N) information. In Fig. 8(b), however, the IBD has all of the structural information, in the labels on the interactions between components B_1 and B_2. In the general case, complex systems will treat some interactions as internal, and others as external. But all of the structural information will be found within some diagram, so the full structural information of the interaction table can be constructed. Then, following the procedure presented in Section IV-C, that table can be compared to the GIT derived from the SD (Fig. 7(a), in this example).

The PkD, PmD, and UCD are the remaining SysML diagrams to consider. They are either high-level diagrams (PkD and UCD) or low-level constraints (PmD). They therefore do not have full-system descriptions that would require the analysis presented in this paper to check for consistency. Instead, a very simple analysis can determine whether the names referred to in these diagrams match to the names used in the other diagrams. And to determine whether the dependencies match to the other diagrams. For example, if two blocks belong to two different packages, and the BDD shows an interface, then the two packages must have a dependency relationship. If a parametric constraint involves two blocks then there must be a relationship between them in the IBD or BDD. Our guarded interaction table (GIT) contains all the information need to test the requirements in these remaining SysML diagrams. A caveat is that parametric constraints could be inconsistent with either SD timing information or with the guard condition values, but this is usually not covered in the field of consistency checking.

Table 1 summarizes the above discussion. Some of the items in the table are written in bold face. These are the items that occur in more than one row of the table. The significance of the same information being present in two or more diagrams is that it creates the possibility that that information could be inconsistent between those diagrams. Conversely any text not in bold face represents information that is only present in one diagram—which means that that information is irrelevant to the current problem of consistency checking between diagrams. In some cases, inconsistency checking will not require the techniques of this present paper because the potential inconsistencies are trivial to identify (e.g., the AD's behavior implies a structure where a specific object will have been defined to perform the behavior). These trivial cases are not shown in bold, even if they occur in multiple rows.

Table 1. A summary of the structural and behavioral information found within the various SysML diagrams

Diagram type	Structural information	Behavioral information
BDD	**Objects** (i.e., blocks), **Operational signatures** (external)	N/A
IBD	**Objects** (i.e., parts), **Operational signatures** (internal)	N/A
SD	**Objects**, **Operational signatures**, Timing requirements	**Composition of processes:** - **Strict sequential** - Weak sequential - **Parallel** - **Alternative** Message timings
SMD	**Objects** (for class-level SMDs) or Implied objects (otherwise)	**Composition of processes:** - **Strict sequential** - **Parallel** - **Alternative**
AD	**Objects** (if swimlanes are used) or Implied objects (otherwise)	**Composition of processes:** - **Strict sequential** - **Parallel** - **Alternative**
PkD, PmD, UCD	Lack full-system descriptions needed for detailed consistency checking. Yet block names and dependencies do get checked.	

Some items are written in bold face, indicating that these items occur in multiple rows. Only when such information is present in two or more diagrams does the possibility exist for inconsistencies between those diagrams.

Looking at the table, the sequence diagram is found to contain all of the bold text, meaning that we can know that all the diagrams are consistent with each other by simply showing that each diagram is individually consistent with the sequence diagram (But this does not identify which diagram has the error, if an inconsistency is found.).

C. Consistency Checking Using the ITG and GIT

Checking for behavioral consistency between diagrams means determining if the ITGs derived from those diagrams are bisimilar. The benefit of using ITGs created from process algebra equations is that the well-known formalism of process algebra can be leveraged for bisimilarity testing. In fact, this issue has been discussed in our previous paper on behavioral consistency checking between SMDs and SDs [4].

One complication is that some system engineers may not have designed all diagrams at the same level of detail. In particular, the SMD might be used to describe the behavior only of single object classes, not the high-level interactions between those objects. The only inconsistencies that such a class-level SMD can have with other diagrams will be in regards to the behavior of that class of object. Consequently, one way to check consistency would be to perform multiple smaller-sized consistency checks on just those portions of the ITG that pertain to those specific objects.

A more elegant solution that we propose is to reuse the upper portion of the SD's ITG, in order to allow a full-system ITG to be created from class-based SMDs. The reason the SD's ITG is the one that is chosen to fill in the gaps of the SMDs is because the SD is SysML's most complete diagram. Moreover, by reusing information from the SD's ITG, there will be no possibility of introducing any new inconsistencies to the parts of the ITG that were derived from the individual SMDs of the different objects in the system.

A further advantage of the proposed solution can be noted. Since the ITG's format is consistent with the rules for describing a SysML SMD, our approach will allow users of class-level SMDs to also see how the higher-level system behavior is represented in their familiar SMD format. In fact, this idea can even be taken to its extreme: system designers can use the ITG derived from the SD to be their SMD. That is: they do not need to create the SMD by hand, because removing the ITG's structural information fields will allow for the automatic generation of the SMD.

Whereas behavioral inconsistencies can be identified by checking ITGs for bisimilarity, structural inconsistencies are identified by checking the GIT for compatibility with the BDD/IBD. The procedure involves three steps: (1) extracting an interaction table from the IBD and BDDs, (2) reducing the GIT to a similar table, and (3) checking the reduced GIT table for one-to-oneness with the IBD/BDD-derived table.

First, extracting an interaction table from the IBD and BDDs can be understood by considering the example Fig. 8. Now, in the case of Fig. 8(b), none of the relevant structural information is in the BDDs (since user interactions were not specified). Consequently, the structural information is trivial to extract from the interaction edge labels of the IBD.

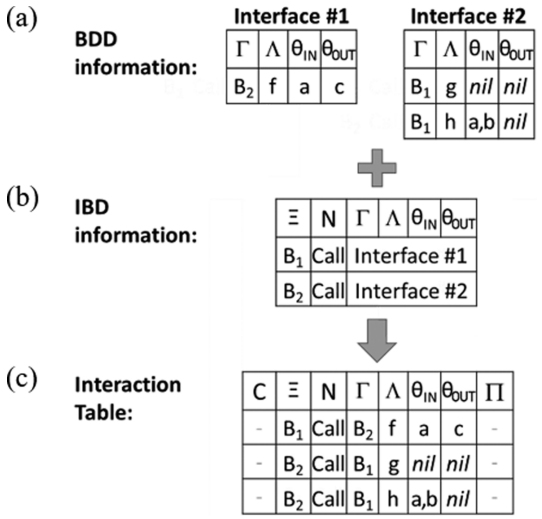


Fig. 9. Extracting the interaction table from the BDD and IBD. The above is for the specific BDDs and IBD of Fig. 8(a). As shown, the BDD contains information about the interface (fields Γ, Λ, θ_{IN}, and θ_{OUT}). The IBD determines how these interfaces are accessed (fields Ξ and N). The C and Π fields of the GIT hold behavioral information that cannot be obtained from the BBD or IBD.

The more interesting case is when structural information is present in both diagram types, as in Fig. 8(a). In this case, the procedure to create the interaction table is illustrated in Fig. 9. As the figure shows, small interface tables were made from the BDDs of Fig. 8(a). Comparing these tables to the BDDs in Fig. 8(a), the Γ field comes from the Ports-and- Interfaces BDD, while the Λ, θ_{IN}, θ_{OUT} fields come from the Interfaces BDDs. As Fig. 9 further shows, another table has been created for the IBD, with values for the Ξ and N fields. Fig. 9 finally combines the above small tables into the full interaction table. This interaction table contains all of the structural information of the system, but it does not contain behavioral information, so the C and Π fields of the GIT are left blank.

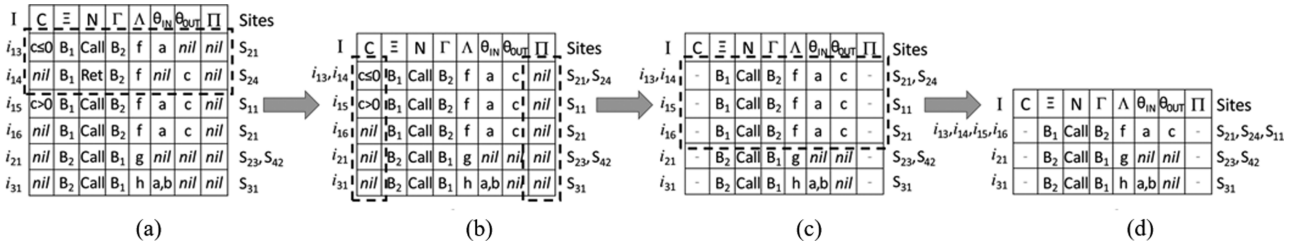


Fig. 10. Reducing the GIT derived from the SD. In (a), the GIT from Fig. 7(a) has been reproduced, but with a dashed box around the rows for i_{13} and i_{14}. This box indicates that these rows will be merged. In (b), rows i_{13} and i_{14} have been merged (meaning that f()'s return rolled into its call). Dashed boxes now indicate that the structural information in the C and Π fields is to be removed. In (c), a new dashed box notes that three of the rows are now identical, and can be merged. In (d), the final interaction table is obtained, containing all of the structural information from the SD.

Second, reducing the GIT to a similar table is illustrated in Fig. 10. The reduction from Fig. 10(a) to Fig. 10(b) involves merging f()'s call and return. This merging may be confusing, because the resulting *N* field retains the string "Call" from the i_{13} row. The reason that the string did not change is because "Call" can refer to either a call or a call-and-return, with these two cases being distinguished by whether there is a value in the θ_{OUT} field. In the merged row of Fig. 10(b), the θ_{OUT} field's value has come from Fig. 10(a)'s i_{14} row. Furthermore, it should be clarified that the reason for rolling f()'s return into its call is because it was for behavioral reasons that they were initially separated. Referring back to the SD shown in Fig. 6(a), three f() operations occur, but only the third one had its call (l_{13}) and return (l_{14}) separated by other system behaviors. Therefore, in Fig. 7(a), the other two f() operations were already shown with their returns merged into their calls.

Fig. 10(c) further reduces the table by removing the behavioral information in the *C* and Π columns, as this information cannot create inconsistencies with the structural BDD and IBD diagrams. Fig. 10(c) also indicates that three of the rows in the table are now identical, because they only differed in behavior. It is legal to merge identical rows, because they convey the same requirements for structural consistency checking.

Fig. 10(d) contains the final interaction table. Comparing to the table in Fig. 9(c), they are identical, which proves structural consistency between the SD of Fig. 4 and the IBD/BBDs of Fig. 8.

Moving beyond the above example, the rule for the general case is not that these two tables be identical; it is that the reduced table for the SD's GIT (as in Fig. 10(d)) must have a one-to-one (but not necessarily onto) mapping to the table produced from the IBD/BBDs (as in Fig. 9(c)). The reason for this requirement is analogous to a requirement that compilers of object-oriented languages have: it is not required that a program must use all of the methods defined by a class it uses, but it is required that the class must define all of the methods that the program does use.

V. A TOOL FOR DIAGRAM SPECIFICATION AND CONSISTENCY CHECKING

To support this research, a tool has been developed both to allow process algebra equations to be transformed into different SysML diagrams, and also to enable consistency checking between diagrams.

Consistency checking is facilitated by the above-described procedure to represent all of the structural and behavioral information contained within each of the various SysML diagram types by using one unified format: the ITG. The ITG follows the SBC approach of structure-behavior coalescence, with a system's structural information being contained within the interactions that serve as the labels of the transition edges between the states of the ITG. The GIT is constructed by collecting all of these interactions into one table.

To practically implement these concepts, a new file

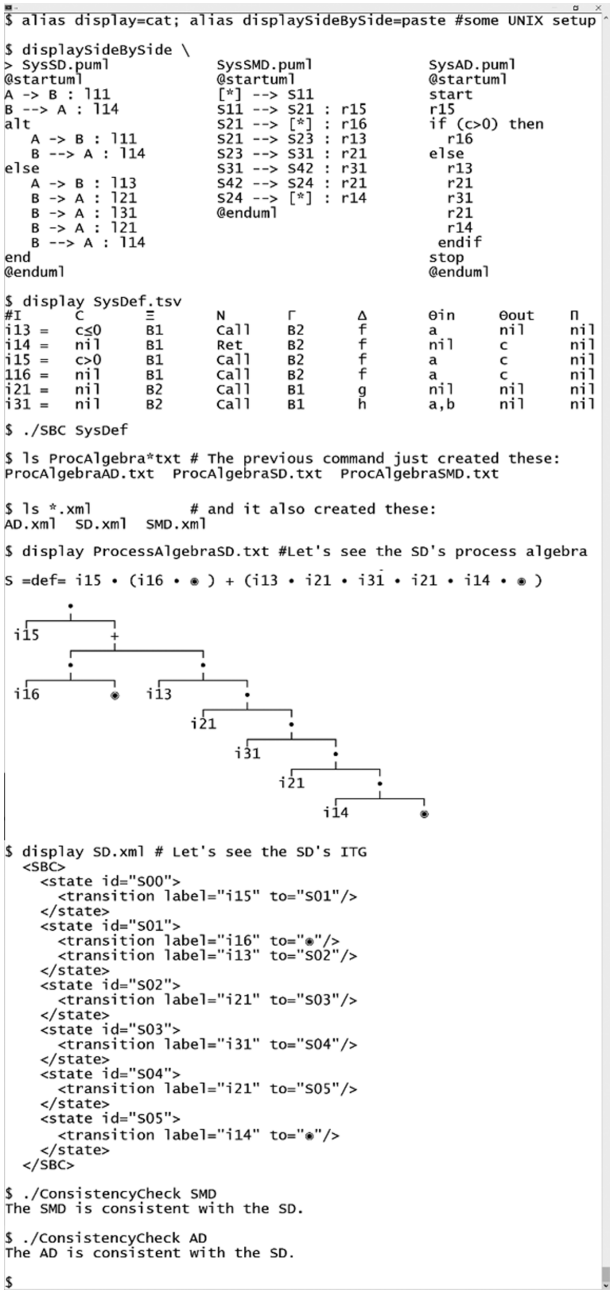


Fig. 11. Tool demonstration for behavior. In this example, the tool is run in UNIX, with "$" as the command line indicator symbol. At the top, a line defines aliases to display files (how the aliases work is unimportant). Next shown are the specifications for: the SD of Fig. 4(a), the SMD of Fig. 5(c), and the AD of Fig. 5(a), using the puml format[ref]. Next, the user-provided SysDef.tsv file shows how interactions are specified. Next, our SBC tool is executed. The "ls" commands then show the txt and xml files that were created by running the SBC tool. Taking the SD, its process algebra (with corresponding parse tree) and its ITG are shown. Next, consistency checking is performed between the three diagrams.

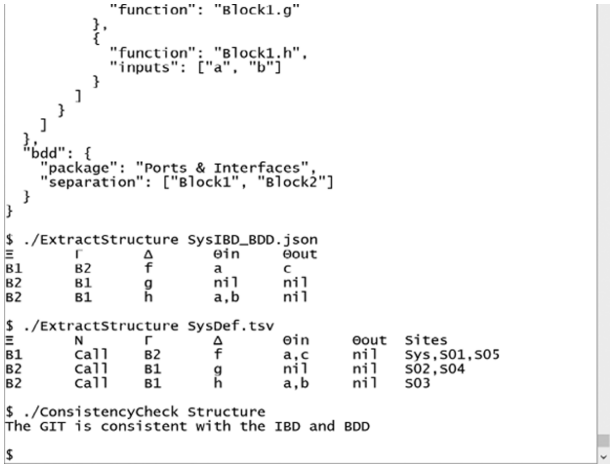


Fig. 12. Tool demonstration for structure. Continuing this example, the IBD and BDD of Fig. 8(b) are shown. Next, our ExtractStructure tool is executed twice. For the IBD and BDD, the extraction result is the table of Fig. 9(c). For the SysDef.tsv file, however, the result is the GIT of Fig. 10(d), which retains behavioral information with the structural information. Finally, consistency checking for structure finds no problems, as all of the functions that can be invoked in this system's behavior are properly defined in the IBD/BDD structure.

format was developed for representing this structural and behavioral information. An XML-based file format was chosen, because it is a format often used for state machine representations, and because it is a flexible format. The format is flexible because it only specifies the general structure of the states, leaving the designer to specify the full format.

An example will now demonstrate our chosen format. Fig. 11 shows the use of our tool to convert the process algebra equation shown in Fig. 7(a) into the ITG shown in Fig. 7(b). As Fig. 11 shows, an input file of process algebra equations is provided, along with definitions for the interactions of those equations. That input file is provided to our tool, which creates the XML output file. In the top portion of the file, the ITG is described. To understand this XML representation, compare it to its graphical representation in Fig. 7(b). The bottom portion of the XML file presents the GIT. This defines the labels of the transition edges of the ITG from the top of the XML file.

Fig. 12 presents the interactions extracted from the IBD and BDD of Fig. 9. It also demonstrates using our tool for consistency checking between that extracted information and the GIT found in the XML file. This verifies that the interactions specified in the behavioral diagrams are properly achievable within the specified structure of the system.

VI. CONCLUSION

By incorporating SBC, this approach ensures that both the structure and behavior of a system are considered in tandem, offering a comprehensive view of the system's design. The ITG model integrates information from various SysML diagrams, including SD, SMD, and AD, into a single consistent format that facilitates straightforward consistency checking.

A software tool is also presented, which demonstrates the practical application of this approach, automatically converting SysML diagrams into the ITG format and performing consistency checks. By employing formal process algebra, the tool allows for an efficient analysis of complex systems, ensuring that all structural and behavioral components are aligned. Future work will explore expanding the tool's capabilities. Moreover, the potential for applying this approach to other modeling languages, such as UML, or adapting it for more dynamic systems remains a promising avenue for research.

ACKNOWLEDGEMENT

This study was supported by a grant from the National Science and Technology Council of Taiwan (NSTC 113-2221-E-110-080).

CONFLICT OF INTEREST

The authors have declared that no competing interests exist.

REFERENCES

1. S. Haga, W. M. Ma, and W. S. Chao, "Structure-behavior coalescence method for formal specification of UML 2.0 sequence diagrams," *Journal of Computing Science and Engineering*, vol. 15, no. 4, pp. 148-159, 2021. http://dx.doi.org/10.5626/JCSE.2021.15.4.148
2. S. W. Haga, W. M. Ma, and W. S. Chao, "Formalizing UML 2.0 state machines using a structure-behavior coalescence method," in *Proceedings of 2022 IEEE Industrial Electronics and Applications Conference (IEACon)*, Kuala Lumpur, Malaysia, 2022, pp. 174-179. https://doi.org/10.1109/IEACon55029.2022.9951741
3. S. W. Haga, W. M. Ma, and W. S. Chao, "Using the structure-behavior coalescence method to formalize the action flow semantics of UML 2.0 activity diagrams," *Journal of Computing Science and Engineering*, vol. 17, no. 2, pp. 60-70, 2023. http://dx.doi.org/10.5626/JCSE.2023.17.2.60
4. S. W. Haga, W. M. Ma, and W. S. Chao, "Inconsistency checking of UML sequence diagrams and state machines using the structure-behavior coalescence method," in *Proceedings of 2022 International Conference on Engineering and Emerging Technologies (ICEET)*, Kuala Lumpur, Malaysia, 2022, pp. 1-6. https://doi.org/10.1109/ICEET56468.2022.10007160
5. D. Torre, Y. Labiche, M. Genero, and M. Elaasar, "A sys-

tematic identification of consistency rules for UML diagrams," *Journal of Systems and Software*, vol. 144, pp. 121-142, 2018. https://doi.org/10.1016/j.jss.2018.06.029
6. D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*, 2nd ed. Upper Saddle River, NJ: Addison-Wesley, 2009.
7. J. A. Estefan, "Survey of model-based systems engineering (MBSE) methodologies," INCOSE MBSE Initiative, 2008 [Online]. Available: https://www.researchgate.net/publication/237302799.
8. N. M. Minhas, A. M. Qazi, S. Shahzadi, and S. Ghafoor, "An Integration of UML sequence diagram with formal specification methods: a formal solution based on Z," *Journal of Software Engineering and Applications*, vol. 8, no. 8, pp. 372-383, 2015. https://doi.org/10.4236/jsea.2015.88037
9. A. Knapp, T. Mossakowski, M. Roggenbach, and M. Glauer, "An institution for simple UML state machines," in *Fundamental Approaches to Software Engineering*. Heidelberg, Germany: Springer, 2015, pp. 3-18. https://doi.org/10.1007/978-3-662-46675-9_1
10. H. Fecher and J. Schonborn, "UML 2.0 state machines: complete formal semantics via core state machine," in *Formal Methods: Applications and Technology*. Heidelberg, Germany: Springer, 2006, pp. 244-260. https://doi.org/10.1007/978-3-540-70952-7_16
11. S. Bernardi, S. Donatelli, and J. Merseguer, "From UML sequence diagrams and statecharts to analysable petri net models," in *Proceedings of the 3rd International Workshop on Software and Performance*, Rome, Italy, 2002, pp. 35-45. https://doi.org/10.1145/584369.584376
12. D. Allaki, M. Dahchour, and A. En-Nouaary, "Managing inconsistencies in UML models: a systematic literature review," *Journal of Software*, vol. 12, no. 6, pp. 454-471, 2017. https://doi.org/10.17706/jsw.12.6.454-471
13. J. Magee and J. Kramer, *Concurrency State Models and Java Programs*, 2nd ed. Hoboken, NJ: John Wiley & Sons, 2006.
14. R. R. Chen, C. C. Lin, L. Wang, and W. S. Chao, "Software architecture design of animation studies platform using structure-behavior coalescence method," *Journal of Circuits, Systems and Computers*, vol. 31, no. 6, article no. 2250103, 2022. https://doi.org/10.1142/S0218126622501031



Steve W. Haga https://orcid.org/0000-0002-9199-2564

Steve W. Haga received Ph.D. degree in Electrical Engineering from the University of Mayland, College Park, USA in 2005. He joined the Computer Science and Engineering at National Sun Yat-sen University as an assistant professor in Taiwan in 2012. His research interests are system modeling and AI-based education.



William S. Chao https://orcid.org/0000-0002-5293-5639

William S. Chao received Ph.D. degree in information science from the University of Alabama at Birmingham, USA in 1988. He joined in the GE Research and Development Center, Schenectady, New York, USA in December 1988. Dr. Chao also works for National Sun Yat-Sen University, Taiwan. His research interests are SBC process algebra, SBC systems architecture, SBC software architecture, SBC-trained AI.



Wei-Ming Ma https://orcid.org/0000-0002-9334-1415

Wei-Ming Ma received Ph.D. degree in Physical Oceanography from the Florida Institute of Technology, FL, USA, in 1997. He joined the Department of Information Management at Cheng-Shiu University as a professor in 2006. His research interests are SBC architecture, enterprise architecture, artificial intelligence, information security, digital forensics, and AR/VR multimedia Design.