Model Construction and Priority Synthesis for Simple Interaction Systems

Chih-Hong Cheng[1], Saddek Bensalem[2], Barbara Jobstmann[2], Rongjie Yan[3], Alois Knoll[1], and Harald Ruess[4]

[1] Department of Informatics, Technischen Universität München, Germany
[2] Verimag Laboratory, Grenoble, France
[3] State Key Laboratory of Computer Science, Institute of Software, CAS, China
[4] Fortiss GmbH, Munich, Germany

Abstract. VissBIP is a software tool for visualizing and automatically orchestrating component-based systems consisting of a set of components and their possible interactions. The graphical interface of VissBIP allows the user to interactively construct BIP models [3], from which executable code (C/C++) is generated. The main contribution of VissBIP is an analysis and synthesis engine for orchestrating components. Given a set of BIP components together with their possible interactions and a safety property, the VissBIP synthesis engine restricts the set of possible interactions in order to rule out unsafe states. The synthesis engine of VissBIP is based on automata-based (game-theoretic) notions. It checks if the system satisfies a given safety property. If the check fails, the tool automatically generates additional constraints on the interactions that ensure the desired property. The generated constraints define priorities between interactions and are therefore well-suited for conflict resolution between components.

1 Introduction

We present VissBIP[1], an open-source tool to construct, analyze, and synthesize component-based systems. Component-based systems can be modeled using three ingredients: (a) *Behaviors*, which define for each basic component a finite set of labeled transitions (i.e., an automaton), (b) *Interactions*, which define synchronizations between two or more transitions of different components, and (c) *Priorities*, which are used to choose between possible interactions [3].

In the BIP framework [3], the user writes a model using a programming language based on the Behavior-Interaction-Priority principle. Using the BIP toolset, this model can be compiled to run on a dedicated hardware platforms. The core of the execution is the *BIP engine*, which decides which interactions are executed and ensures that the execution follows the semantics.

[1] VissBIP is a shortcut for **Vi**sualization and **S**ynthesis of **S**imple **BIP** models. It is available at `http://www6.in.tum.de/~chengch/vissbip`

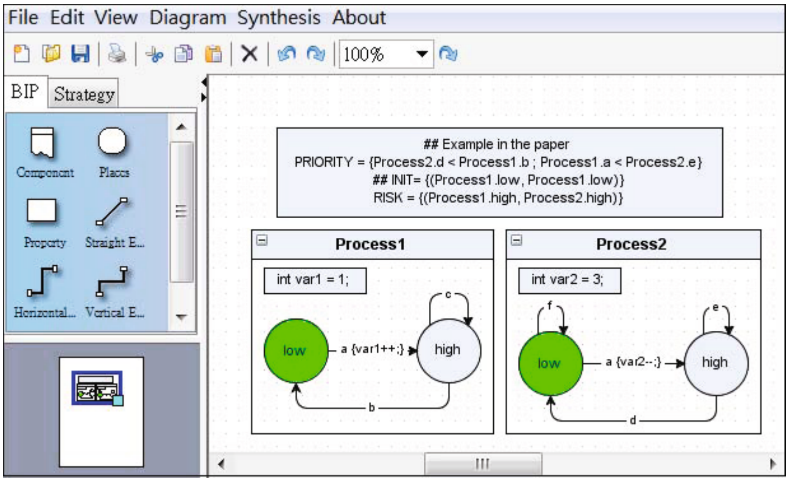
VissBIP is a tool for constructing and visualizing BIP models. Its graphical interface allows the user to model hierarchical systems. The analysis and synthesis engine can currently only interpret non-hierarchical model, which we call *simple.* In BIP, a system is built by constructing a set of basic components and composing them using interactions and priorities. The interactions and priorities are used to ensure global properties of the systems. For instance, a commonly seen problem is mutual exclusion, i.e., two components should avoid being in two dedicated states at the same time. Intuitively, we can enforce this property by requiring that interactions that exit one of the dedicated states have higher priority than interactions that enter the states. Adding interactions or priorities to ensure a desired behavior of the overall systems is often a non-trivial task. VissBIP supports this step by automatically adding a set of priorities that enforce a desired safety property of the composed systems. We call this technique **priority synthesis**. We concentrate on adding priorities because (1) priorities preserve already established safety properties as well as deadlock-freedom of the system, and (2) priorities can be implemented efficiently by allowing the components to coordinate temporarily [6].

2 Visualizing Simple Interaction Systems

The user can construct a system using the drag-and-drop functionality of VissBIP's graphical user interface shown in Figure 1. BIP objects (components, places, properties, and edges) can be simply dragged from the menu on the left to the drawing window on the right.

We use the system shown in Figure 1 to illustrate how a system is represented. The system consisting of two components (`Process1` and `Process2`) depict as boxes. Each component has two places (`high` and `low`) and a local variable (`var1` and `var2`, respectively). A place (also called location) is represented by a circle. A green circle indicates that this place is an initial location of a behavioral component. E.g., place `v1` is marked as initial in `Process1`. Squares denotes variables definitions and their initialization within a component. E.g., `var1` and `var2` are both initialized to 1. Edges between two locations represent *transitions.* Each transition is of the format `{precondition} port-name {postcondition}`. E.g., the transition of `Process1` from place `low` to `high` is labeled with port name `a` and upon its execution the value of `var1` is increased by 1. For simplicity we use **port-name bindings** to construct interactions between components, i.e., that transitions using the same port name are automatically grouped to a single interaction and are executed jointly[2]. In the following, we refer to an interaction by its port name. Finally, additional squares outside of any component, are used to define system properties such as priorities over interactions and winning conditions (for synthesis or verification). In particular, we use the keyword `PRIORITY` to state priorities. E.g., the statement `Process2.d < Process1.b` means that whenever interactions `b` and `d` are available, the BIP engine always

[2] It is possible to pass data through an interaction. The user specifies the data flow associated to an interaction in the same way she describes priorities (see below).



```
## Example in the paper
PRIORITY = {Process2.d < Process1.b ; Process1.a < Process2.e}
## INIT= {(Process1.low, Process1.low)}
RISK = {(Process1.high, Process2.high)}
```

```
int var1 = 1;
```

```
int var2 = 3;
```

Fig. 1. Model construction using VissBIP

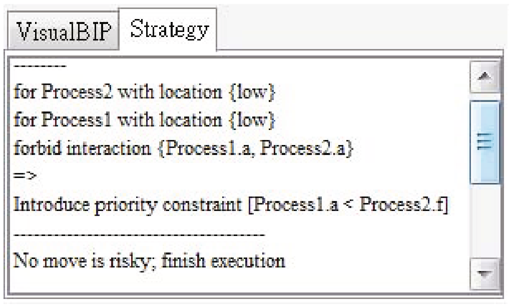
executes `b`. The keyword `RISK` is used to state risk conditions. E.g., the condition `RISK = {(Process1.high, Process2.high)}` states that the combined location pair `(Process1.high, Process2.high)` is never reached. Apart from the stated conditions, we also implicitly require that the system is deadlock-free, i.e., at anytime, at least one interaction is enabled. When only deadlock avoidance is required, the keyword `DEADLOCK` can be used instead. Lines started with `##` are comments.

3 Priority Synthesis

We define priority synthesis as an automatic method to introduce a set of new priorities over interactions on a BIP system such that the augmented system satisfies the specified property. A priority is *static* if it does not contain evaluations over ports or locations in components as a precondition to make the priority active. We focus on synthesizing static priorities. We consider safety (co-reachability) properties, i.e., the property specifies a set of risk states, and the system should never reach any of them. For the rest of this section, we first show the results of priority synthesis under VissBIP. Then, we give some details about the underlying algorithm and the implementation.

3.1 Safety Synthesis by Adding Global Priorities: Examples

The user can invoke the synthesis engine on a system like to one shown in Figure 1. The engine responds in one of the following three ways: (1) It reports that no additional priorities are required. (2) It returns a set priority constraints that ensure the stated property. (3) It states that no solution based on priorities can be found by the engine.



```
--------
for Process2 with location {low}
for Process1 with location {low}
forbid interaction {Process1.a, Process2.a}
=>
Introduce priority constraint [Process1.a < Process2.f]
--------------------------------------
No move is risky; finish execution
```

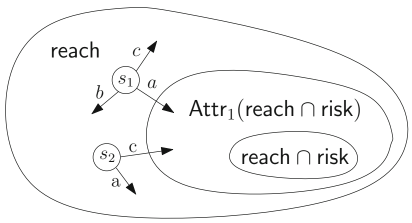

reach
s_1
s_2
a
b
c
c
a
$\mathsf{Attr}_1(\mathsf{reach} \cap \mathsf{risk})$
$\mathsf{reach} \cap \mathsf{risk}$

Fig. 2. The automatically synthesized priority for the model in Figure 1 (left), and an example for priority resolution (right)

Figure 2 shows the *Strategy* panel of VissBIP, which displays the results obtained by invoking the synthesis engine on the example of Figure 1. Recall, that in the example, we stated that the combined location pair `(Process1.high, Process2.high)` is never reached. The engine reports that the priority constraint `Process1.a < Process2.f` should be added. Note that if the system is in state (`Process1.low, Process2.low`), then the interaction `Process1.a` (which is a joint action from Process 1 and Process 2) would immediately leads to a risk state (a state satisfying the risk condition). This can be avoided by executing `Process2.f` first. The new priority ensures that interaction `f` is executed forever, which is also deadlock-free.

3.2 Safety Synthesis by Adding Global Priorities: Algorithmic Issues

The algorithm of priority synthesis is based on concepts in games [5]. A game is a graph partitioned into player-0 and player-1 vertices. We refer to Player 0 as the system (which is controllable) and Player 1 as the environment (which is uncontrollable). In controllable vertices, the system can choose among the set of available transitions during execution. Conceptually, a play between two players is proceeded as follows:

1. A player-1 vertex is a product of (i) locations of behavioral components and (ii) evaluations of their variables. Since the values of the variables are not relevant in our example, we omit them for simplicity. E.g., in Figure 1, (`Process1.low, Process2.low`) is a player-1 vertex.
2. From a player-1 vertex the game moves to a player-0 vertex that represents all the available interactions. E.g., from location (`Process1.low,Process2.low`), the game proceeds to state labeled with the interactions a and f.
3. Then, Player 0 responds by selecting one interaction and updates the location (i.e., to a new vertex of player-1). Note that its selection is constrained by the pairing of interactions as well as the priority specified in the system.

Admittedly now in our formulation player-1 is deterministic (thus it can be viewed as 1-player game, or automaton). Nevertheless, a non-deterministic player-1 is introduced when data is considered using abstraction techniques. The algorithmic flow for priority synthesis is as follows.

- (Game construction) First, VissBIP creates a symbolic representation of the game specified above using the BDD package JDD [2]. Then, the engine compute the set of reachable risk states by intersecting the set of reachable states reach with the set of risk states risk derived from the specification. In order to obtain a good initial variable ordering, we use heuristics to keep variables from component that participate in the same interaction close.
- (Game solving and risk-edge generation) Once the symbolic representation of the game is constructed, VissBIP solves the safety game by symbolically computing the *risk attractor* $\mathsf{Attr}_1(\mathsf{reach} \cap \mathsf{risk})$, which is the set of states from which player-1 can force to move to a risk state, regardless of moves done by Player 0 [5]. If all the reachable moves of Player 0 avoid the risk attractor, then the model running on the BIP engine is guaranteed to be safe. Otherwise, we derive a set of *risk edges*, which are all the edges leading to the risk attractor. E.g., Figure 2 shows that from (`Process1.low, Process2.low`) interaction `a` corresponds to a risk edge. We compute the set of risk edges T_{risk} with the following formula: $(\mathsf{reach} \setminus \mathsf{Attr}_1(\mathsf{reach} \cap \mathsf{risk})) \cap T_0^\sharp \cap \mathsf{Attr}'_1(\mathsf{reach} \cap \mathsf{risk})$, where $\mathsf{Attr}'_1(\mathsf{reach} \cap \mathsf{risk})$ is the primed version of $\mathsf{Attr}_1(\mathsf{reach} \cap \mathsf{risk})$.
- (Risk-edge interpretation) We aim to introduce priority constraints to prevent the BIP engine from selecting transitions in T_{risk}. This can be done by examining interactions that are also available at the locations from which a risk edge can be taken. E.g., in our example (Figure 1), the engine examines all alternative interactions at state (`Process1.low`, `Process2.low`). Since `Process2.f` can also be selected, the engine generates the priority constraint `Process1.a < Process2.f` to avoid using `Process1.a`.
  To avoid enumerating all the risk edges, VissBIP aims to rule out risk-edges in a symbolic fashion. More precisely, VissBIP proceeds on cubes of the risk-edges, which are sets of edges that can be represented by a conjunction over the state variables or their negation. For each cube, the engine generates a set of candidate priorities that can be used to avoid these risk edges.
- (Priority generation) When the engine has collected the set of priorities for each cube in T_{risk}, these priorities are as priority fixes having preconditions (E.g., in Figure 2 if on state s_2 we should use priority $c < a$ to escape from the attractor). We are interested in adding static priorities, which are independent of the actual state. VissBIP offers the user to select between two incomplete algorithms to obtain static priorities.
  - *Priority resolution using SAT solvers*: From the set of candidate priorities obtained for each cube, the engine needs to select a set of non-conflicting priorities. E.g., consider the example in Figure 2, for state s_1 VissBIP generates a candidate set $\{(a < b), (a < c)\}$ and for s_2 it gives $\{(c < a)\}$. The engine should not report $\{(a < c), (c < a)\}$ as a static priority fix, as it does not satisfy the strict partial order. Finding a set

of priorities satisfying each cube while obeying the strict partial order can be done using SAT solvers. In VissBIP we use SAT4J [1].

– *Fixed-size priority selection*: For this scheme, the engine examines all possible subsets of the collected priorities with the size bounded by a user-specified number, starting from the smallest subset. Since in general a small number of modification of the system is desirable, this is a natural approach. Furthermore, it gives appealing results on our examples.

4 Evaluation and Summary

In this paper, we present a tool for constructing simple BIP systems together with a technique called priority synthesis, which automatically adds priorities over interactions to maintain system safety and deadlock-freedom.

We have evaluated the tool on some examples, e.g., traditional dining philosophers problems, and problems related to critical region control. On these examples, VissBIP enables to generate small yet interesting priority fixes. Due to the limitation concerning the number of components to be placed in a single canvas, examples under investigations are admittedly not very big, but they can be solved within reasonable time. E.g., the dining philosophers problem with size 12 (i.e., a total of 24 interacting components) is solved within 1 seconds using the SAT-resolution method and within 3 seconds using constant-depth fixing[3]. As the fix is automatically generated without human intervention, we treat this as a promising step towards computer-aided synthesis in BIP. Algorithms in VissBIP can be viewed as new features for the future D-Finder tool [4], which focuses on deadlock finding and verification over safety properties. Nevertheless, the front-end GUI targeted for the ease of model construction is also important.

Lastly, as priority synthesis is essence a method of *synthesizing simple component glues for conflict resolution*, under suitable modifications, our technique is applicable for multicore/manycore systems working on task models for resource protection, which is our next step.

References

1. SAT4J: a SAT-solver based on Java, `http://www.sat4j.org`
2. The JDD project, `http://javaddlib.sourceforge.net/jdd/`
3. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in bip. In: SEFM 2006, pp. 3–12. IEEE, Los Alamitos (2006)
4. Bensalem, S., Bozga, M., Nguyen, T., Sifakis, J.: D-finder: A tool for compositional deadlock detection and verification. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 614–619. Springer, Heidelberg (2009)
5. Grädel, E., Thomas, W., Wilke, T.: Automata, Logics, and Infinite Games. LNCS, vol. 2500. Springer, Heidelberg (2002)
6. Graf, S., Peled, D., Quinton, S.: Achieving distributed control through model checking. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 396–409. Springer, Heidelberg (2010)

[3] The result is evaluated under the Java Virtual Machine with parameter `-Xms1024m -Xmx2048m`. The system runs under Linux with an Intel 2.8 Ghz CPU.